CV184X 外围设备驱动使用手册

Version: 1.0.0

Release date: 2025-03-12

目录

修订记录

Revision	Date	Description
1.0.0	2025/03/12	Initial version
1.0.1	2025/07/08	Add CVI_PINMUX_Operation_Guide
1.0.2	2026/04/28	Update PWM/ADC/I2C/SPI/GPIO/UART/Watchdog operation guides

1 声明

法律声明

本数据手册包含北京晶视智能科技有限公司（下称“晶视智能”）的保密信息。未经授权，禁止使用或披露本数据手册中包含的信息。如您未经授权披露全部或部分保密信息，导致晶视智能遭受任何损失或损害，您应对因之产生的损失/损害承担责任。

本文件内信息如有更改，恕不另行通知。晶视智能不对使用或依赖本文件所含信息承担任何责任。

本数据手册和本文件所含的所有信息均按“原样”提供，无任何明示、暗示、法定或其他形式的保证。晶视智能特别声明未做任何适销性、非侵权性和特定用途适用性的默示保证，亦对本数据手册所使用、包含或提供的任何第三方的软件不提供任何保证；用户同意仅向该第三方寻求与此相关的任何保证索赔。此外，晶视智能亦不对任何其根据用户规格或符合特定标准或公开讨论而制作的可交付成果承担责任。

联系我们

地址 北京市海淀区丰豪东路 9 号院中关村集成电路设计园（ICPARK）1 号楼

深圳市宝安区福海街道展城社区会展湾云岸广场 T10 栋

电话 +86-10-57590723 +86-10-57590724

邮编 100094（北京）518100（深圳）

官方网站 https://www.sophgo.com/

技术论坛 https://developer.sophgo.com/forum/index.html

2 Ethernet 操作指南

2.1 操作示例

Ethernet 模块默认为编入内核，不需另外执行加载操作。

内核下使用网口的操作步骤如下：

- 配置 ip 地址和子网掩码

```
ifconfig eth0 xxx.xxx.xxx.xxx netmask xxx.xxx.xxx.xxx up
```

- 设置缺省网关

```
route add default gw xxx.xxx.xxx.xxx
```

- 验证网络连通性

```
ping xxx.xxx.xxx.xxx
# ping 网关或目标服务器 IP，确认网络可达后再进行后续 NFS/TFTP 操作
```

- 挂载 nfs

```
mount -t nfs -o nolock xxx.xxx.xxx.xxx:/your/path /mount-dir
```

- shell 下使用 tftp 上传下载文件

  前提是在 server 端有 tftp 服务软件在运行。

  下载文件：

```
tftp -g -r [remote file name] [server ip]
```

  备注：remote file name 为欲下载的文件名称，server ip 为下载文件所在 server 的 ip 地址 (Ex：tftp -g -r test.txt 192.168.0.11)。

  上传文件：

```
tftp -p -l [local file name] [server ip]
```

  备注：local file name 为本地欲上传的文件名称，server ip 为上传文件的目标 server 的 ip 地址 (Ex：tftp -p -l test.txt 192.168.0.11)。

备注说明：CV184x Ethernet 模块不支持 TSO 功能。

备注说明：nfs 工具默认不会编入文件系统，需要用户在需要时候加入。

2.2 IPv6 说明

SDK 包中默认关闭 IPv6 功能。如果要开启 IPv6，需要修改内核选项。具体操作如下：

1) CV184x 系列

修改

build/boards/cv184x/{board_name}/linux/cvitek_{board_name}_defconfig。

Ex: build/boards/cv184x/cv1842hp_wevb_0014a_spinor/linux/

cvitek_cv1842hp_wevb_0014a_spinor_defconfig 新增或修改成 CONFIG_IPV6=y。然后重新编译内核软件。

IPv6 环境配置方法如下：

- 配置 ip 地址以及网关

```
ip -6 addr add <ipv6 address>/ipv6 prefixlen dev <port name>

Ex: ip -6 addr add 2020:abc:102::8888/24 dev eth0
```

- Ping 指定的 IPv6 地址

```
ping -6 <ipv6 address>

Ex: ping -6 2020:abc:102::6666
```

2.3 IEEE 802.3x 流控功能

2.3.1 流控功能描述

CV184x Ethernet 支持 IEEE 802.3x 所定义的流控功能，透过发送流控帧以及接收对端所发送过来的流控帧的方式来达到流控的目的。

- 发送流控帧：

  在接收由对端发送过来的封包的过程中，若发现目前接收端的接收对列可能无法满足接收后续送达的封包时，则本地端会发送流控帧至对端，要求对端暂停一段时间不发送封包，藉此来进行流量控制。

- 接收流控帧：

  当本地端接收到由对端发送过来的流控帧时，本地端会根据帧内的流控时间描述延迟发送封包至对端，等到过了流控延迟时间后，再启动发送。若在等待过程中收到了由对端发送的流控帧其描述的流控时间为 0 时，则会直接启动发送。

2.3.2 流控功能配置

接收流控帧功能默认是关闭的，亦没有提供软件接口配置。

发送流控帧功能的相关文件配置在 linux/drivers/net/ethernet/stmicro/stmmac/stmmac_main.c

```
static int flow_ctrl = FLOW_OFF;
module_param(flow_ctrl, int, 0644);
MODULE_PARM_DESC(flow_ctrl, "Flow control ability [on/off]");
static int pause =PAUSE_TIME;
module_param(pause, int, 0644);
MODULE_PARM_DESC(pause, "Flow Control Pause Time");
```

若欲默认开启流控功能，可修改 flow_ctrl = FLOW_AUTO。

若欲修改默认 pause time，可配置 pause 至目标值。

2.3.3 ethtool 配置接口流控功能

用户可以通过标准 ethtool 工具接口进行流控功能的使能。

ethtool -a eth0 命令查看 eth0 口流控功能状态；打印如下

```
# ethtool -a eth0
Pause parameters for eth0:
Autonegotiate: on
RX: off
TX: off
```

其中，RX 流控是关闭的，TX 流控是关闭的；用户可以通过以下命令打开或关闭 TX 流控：

```
# ethtool -A eth0 tx off（关闭TX流控）
# ethtool -A eth0 tx on（打开TX流控）
```

备注：ethtool 工具默认不会编入文件系统，需要用户在需要时候加入。

3 USB 操作指南

3.1 操作准备

USB 2.0 Host/Device 的操作准备如下:

- U-boot and Linux 内核使用 SDK 发布的 U-boot 与 Kernel。
- 文件系统可以使用本地文件系统 ext4 或 squashfs，也可以使用 NFS。
- Shell script "run_usb.sh" . run_usb.sh 使用内核的 USB ConfigFS 功能来客制化 USB device 装置. 使用者可参考并修改 run_usb.sh 来变更 PID/VID 与 function 的相关参数. 详细操作可参考内核文件" linux/Documentation/usb/gadget_configfs.txt"。

3.2 Uboot 操作过程

3.2.1 Uboot 下 USB Host 配置

Uboot 下只支持 U 盘和硬盘储存设备. USB host 在 uboot 默认为关闭，须打开相关 config。

步骤 1. 打开 uboot 下 USB 相关的驱动:

在板级 U-Boot 配置文件中使能下列选项。配置文件路径一般为 build/boards/cv184x/板名/u-boot/cvitek_板名_defconfig 。请将路径中的板名替换为实际板级目录名（例如 cv1842hp_wevb_0014a_emmc），具体以 SDK 为准。

```
CONFIG_USB=y
CONFIG_DM_USB=y
CONFIG_USB_STORAGE=y
CONFIG_CMD_USB=y
```

若需更详细的 USB DWC2 调试日志（可选），可在同一 defconfig 中增加:

```
CONFIG_USB_DWC2_VERBOSE_DEBUG=y
```

3.2.2 Uboot 下 U 盘

启动 Uboot USB Host 前的准备：

Uboot 下 USB Host 不支持热插入，须在启动 USB host 前先插上设备. 若平台上有 USB Hub，须确认 Hub 电源已打开，USB 路径的 Switch 切到 Host connector。

以 cv184x 为例（对应方法同样适用于 cv181x)：

平台上电，进入 uboot 命令行，输入命令: usb start，观察是否识别成功。

```
cv184x# usb start
starting USB...
Bus usb@04340000: USB DWC2
scanning bus usb@04340000 for devices... 3 USB Device(s) found
       scanning usb for storage devices... 1 Storage Device(s) found
```

若 usb start 后出现枚举报错或无法检测到设备，可在 uboot 命令行执行 setenv usb_pgood_delay XXX，可针对预热较慢或是中间串了 Hub 的设备调整 timeout 值，建议取值范围为 1000-3000。

识别完成后，再输入命令: usb tree，查看识别拓扑与速率。以下举例 USB host 串接 Hub 与 Mass Storage 设备。

```
cv184x# usb tree
USB device tree:
  1  Hub (480 Mb/s, 0mA)
  |   U-Boot Root Hub
  |
  +-2  Hub (480 Mb/s, 100mA)
    |
    +-3  Mass Storage (480 Mb/s, 500mA)
        Prolific Technology Inc. USB SD Card Reader      ABCDEF0123456789AB
```

也可执行 usb storage 查看当前识别的 USB 存储设备摘要：

```
cv184x# usb storage
  Device 0: Vendor:          Rev: 1.00 Prod: SD Card Reader
            Type: Removable Hard Disk
            Capacity: 960.0 MB = 0.9 GB (1966080 x 512)
```

初始化及应用：

识别完成后可进入以下操作。

步骤 1. 查看设备信息

- 命令行执行: usb info 可列出控制器上所有设备；执行 usb info [dev] 可查看指定设备号详情。

```
cv184x# usb info
1: Hub,  USB Revision 1.10
 -  U-Boot Root Hub
 - Class: Hub
 - PacketSize: 8  Configurations: 1
 - Vendor: 0x0000  Product 0x0000 Version 0.0
   Configuration: 1
```

(下页继续)

(续上页)

```
  - Interfaces: 1 Self Powered 0mA
    Interface: 0
    - Alternate Setting 0, Endpoints: 1
    - Class Hub
    - Endpoint 1 In Interrupt MaxPacket 2 Interval 255ms

2: Hub,  USB Revision 2.0
 - Class: Hub
 - PacketSize: 64  Configurations: 1
 - Vendor: 0x058f  Product 0x6254 Version 1.0
   Configuration: 1
   - Interfaces: 1 Self Powered Remote Wakeup 100mA
     Interface: 0
     - Alternate Setting 0, Endpoints: 1
     - Class Hub
     - Endpoint 1 In Interrupt MaxPacket 1 Interval 12ms

3: Mass Storage,  USB Revision 2.10
 - Prolific Technology Inc. USB SD Card Reader      ABCDEF0123456789AB
 - Class: (from Interface) Mass Storage
 - PacketSize: 64  Configurations: 1
 - Vendor: 0x067b  Product 0x2731 Version 1.0
   Configuration: 1
   - Interfaces: 1 Bus Powered 500mA
     Interface: 0
     - Alternate Setting 0, Endpoints: 2
     - Class Mass Storage, Transp. SCSI, Bulk only
     - Endpoint 1 Out Bulk MaxPacket 512
     - Endpoint 2 In Bulk MaxPacket 512
```

步骤 2. FAT 文件系统列举与写入（可选）

- 若 U 盘分区为 FAT，可使用 fatls 列举文件、fatwrite 写入文件。设备与分区号请按实际环境修改（下例为 usb 0 第一个分区 1）。

```
cv184x# fatls usb 0:1 /
          System Volume Information/
  2552840   boot.emmc
  9887872   cfg.emmc
 53395776   data.emmc
   555520   fip.bin
          temp/
   555520   fip_spl.bin
      991   partition_emmc.xml
  3593016   ramboot.itb
  5963904   rootfs.emmc
  4919424   system.emmc
   811136   yoc.bin
  2627628   boot.spinor
  1048704   data.spinor
      959   partition_spinor.xml
  2240512   rootfs.spinor
     7712   run_usb.sh
  1048576   test.bin
```

(下页继续)

(续上页)

```
16 file(s), 2 dir(s)

cv184x# fatwrite usb 0:1 0x80000000 test2.bin 0x100000
1048576 bytes written in 369 ms (2.7 MiB/s)

cv184x# fatls usb 0:1 /
          System Volume Information/
  2552840   boot.emmc
  9887872   cfg.emmc
 53395776   data.emmc
   555520   fip.bin
          temp/
   555520   fip_spl.bin
      991   partition_emmc.xml
  3593016   ramboot.itb
  5963904   rootfs.emmc
  4919424   system.emmc
   811136   yoc.bin
  2627628   boot.spinor
  1048704   data.spinor
      959   partition_spinor.xml
  2240512   rootfs.spinor
     7712   run_usb.sh
  1048576   test.bin
  1048576   test2.bin

17 file(s), 2 dir(s)
```

步骤 3. 对 U 盘进行读操作（FAT）

- 分区为 FAT 时，可使用 fatload 将文件读入内存：fatload usb dev:part loadaddr filename。dev、分区号、loadaddr、filename 请按实际环境修改，示例如下：

```
cv184x# fatload usb 0:1 0x82000000 boot.spinor
2627628 bytes read in 272 ms (9.2 MiB/s)
```

步骤 4. 对 U 盘进行写操作（FAT）

- 分区为 FAT 时，可使用 fatwrite 从内存写入文件：fatwrite usb dev:part addr filename bytes。示例如下（与步骤 2 中 fatwrite 用法一致）：

```
cv184x# fatwrite usb 0:1 0x80000000 test2.bin 0x100000
1048576 bytes written in 369 ms (2.7 MiB/s)
```

3.3 linux Host

3.3.1 USB 2.0 Host 操作过程

步骤 1. 内核配置 使能 USB Host 及 U 盘存储相关配置。配置文件路径一般为 build/boards/cv184x/板名/linux/板名 _defconfig。

```
CONFIG_USB_SUPPORT=y
CONFIG_USB=y
CONFIG_USB_DWC2=y
CONFIG_USB_GADGET=y
CONFIG_USB_CONFIGFS=y
CONFIG_USB_LIBCOMPOSITE=y
CONFIG_USB_ROLE_SWITCH=y
CONFIG_SCSI=y
CONFIG_BLK_DEV_SD=y
CONFIG_USB_STORAGE=y
```

步骤 2. 切换为 Host 模式

若板级通过 GPIO 控制 USB 供电或 Host/Device 切换，请根据硬件设计操作对应 GPIO。本示例中开发板连接 Q3 三极管、断开 Q4 三极管，并连接 R127、R125。例如使用 GPIO 450 打开 USB 供电，请根据实际情况拉高或拉低电平。

```
echo 450 > /sys/class/gpio/export
echo out > /sys/class/gpio/gpio450/direction
echo 1 > /sys/class/gpio/gpio450/value
```

将 OTG 控制器切到 Host 模式:

```
echo host > /proc/cviusb/otg_role
```

步骤 3. 插入 U 盘 将 U 盘插入 USB 口，观察是否枚举成功。正常情况下串口打印类似:

```
[   72.061964] usb 1-1: new high-speed USB device number 2 using dwc2
[   72.315816] usb-storage 1-1:1.0: USB Mass Storage device detected
[   72.335934] scsi host0: usb-storage 1-1:1.0
[   73.363027] scsi 0:0:0:0: Direct-Access     Generic  STORAGE DEVICE   1532 PQ: 0 ANSI: 6
[   73.374407] sd 0:0:0:0: Attached scsi generic sg0 type 0
[   73.558597] sd 0:0:0:0: [sda] 30253056 512-byte logical blocks: (15.5 GB/14.4 GiB)
[   73.566961] sd 0:0:0:0: [sda] Write Protect is off
[   73.571922] sd 0:0:0:0: [sda] Mode Sense: 21 00 00 00
[   73.577899] sd 0:0:0:0: [sda] Write cache: disabled, read cache: enabled, doesn't support DPO␣
↪or FUA
[   73.593961]  sda: sda1
[   73.602607] sd 0:0:0:0: [sda] Attached SCSI removable disk
```

sdXY 中 X 为磁盘号，Y 为分区号，请按实际环境修改。例如 log 中 **sda1** 表示 U 盘第一个分区；多分区时会出现 sda1、sda2、sda3 等。

步骤 4. 查看分区信息 运行 ls /dev 查看设备节点:

- 若无 sdXY（如 sda1），表示尚未分区，需先用 fdisk /dev/sda 分区后再继续。
- 若有 sdXY，表示已识别到 U 盘分区，可进行挂载与读写。

常用命令说明:

- 分区操作设备节点为 sdX，示例: fdisk /dev/sda
- 格式化分区 sdXY（如 FAT32）: mkdosfs -F 32 /dev/sda1
- 挂载分区 sdXY: mount /dev/sda1 /mnt

步骤 5. 挂载 U 盘 挂载目标分区到目录（以 sda1 挂到 /mnt 为例）:

```
mount /dev/sda1 /mnt
```

步骤 6. 对 U 盘进行读写操作 挂载后即可按普通目录读写，例如:

```
# 查看 U 盘内容
ls /mnt

# 从 U 盘复制到当前目录（读）
cp /mnt/auto.sh ./

# 从当前目录复制到 U 盘（写）
cp ./data.bin /mnt/

# 使用完毕后同步并卸载
sync
umount /mnt
```

3.4 linux Device

3.4.1 操作准备

使用 USB Device 前，需在设备树中确保 USB 控制器为 OTG 模式。在板级 DTS 文件中通过覆盖 usb 节点，将 dr_mode 设置为 "otg"。

DTS 路径一般为 build/boards/cv184x/板名/dts_arm/板名.dts。在文件中添加或确认如下覆盖节点:

```
&usb {
    dr_mode = "otg";
};
```

修改后重新编译设备树并烧录至开发板。

3.4.2 切换 switch GPIO

对于 EVB，会通过 switch IC 切换 Host 与 Device 通道，使用 USB 前需先操作对应 GPIO。请根据实际情况拉高或拉低电平。

```
echo 450 > /sys/class/gpio/export
echo out > /sys/class/gpio/gpio450/direction
echo 0 > /sys/class/gpio/gpio450/value
```

3.4.3 Device 作为存储设备的操作指南

1. 内核配置使能 USB Device 及 Mass Storage 相关配置，例如:

```
CONFIG_USB_SUPPORT=y
CONFIG_USB=y
CONFIG_USB_DWC2=y
CONFIG_USB_GADGET=y
CONFIG_USB_CONFIGFS=y
CONFIG_USB_LIBCOMPOSITE=y
CONFIG_USB_ROLE_SWITCH=y
CONFIG_USB_F_MASS_STORAGE=y
CONFIG_USB_CONFIGFS_MASS_STORAGE=y
```

2. 切换为 Device 模式
   - 参考本章 **切换 switch GPIO** 小节操作对应 GPIO。
   - 将 OTG 控制器切到 Device 模式:

```
echo device > /proc/cviusb/otg_role
```

3. 运行脚本 run_usb.sh 脚本位于板子 /etc 目录下:

```
/etc/run_usb.sh probe msc /dev/mmcblk0p1
/etc/run_usb.sh start
```

   mmcblkXY 表示第 X 个磁盘（eMMC 或 SD）的第 Y 个分区，请根据实际情况选择。

4. 连接 USB 线通过 USB 线将平台与 Host 相连，Host 即可将平台识别为 USB 存储设备，并在 /dev 下生成对应设备节点。

3.4.4 Device 作为终端设备的操作指南

1. 内核配置使能 USB Device 及 ACM/Serial 相关配置，例如:

```
CONFIG_USB_SUPPORT=y
CONFIG_USB=y
CONFIG_USB_DWC2=y
CONFIG_USB_GADGET=y
CONFIG_USB_CONFIGFS=y
```

（下页继续）

(续上页)

```
CONFIG_USB_LIBCOMPOSITE=y
CONFIG_USB_ROLE_SWITCH=y
CONFIG_USB_F_ACM=y
CONFIG_USB_CONFIGFS_ACM=y
CONFIG_USB_G_SERIAL=y
CONFIG_USB_U_SERIAL=y
CONFIG_USB_F_SERIAL=y
CONFIG_USB_CONFIGFS_SERIAL=y
```

2. 切换为 Device 模式
   - 参考本章 **切换 switch GPIO** 小节。
   - 将 OTG 控制器切到 Device 模式:

```
echo device > /proc/cviusb/otg_role
```

3. 运行脚本 run_usb.sh 脚本位于板子 /etc 目录下:

```
/etc/run_usb.sh probe acm
/etc/run_usb.sh start
```

4. 连接 USB 线通过 USB 将平台与 Host 相连，Host 即可识别为 USB 终端设备，并生成设备节点 ttyACMX（X 为同类型终端编号）。Device 端 /dev 下会生成 ttyGSY（Y 为同类型终端编号）。Host 与 Device 可通过终端设备进行数据传输。

3.4.5 Device 作为 ADB 设备的操作指南

1. 内核配置使能 USB Device 及 FunctionFS 相关配置，例如:

```
CONFIG_USB_SUPPORT=y
CONFIG_USB=y
CONFIG_USB_DWC2=y
CONFIG_USB_GADGET=y
CONFIG_USB_CONFIGFS=y
CONFIG_USB_LIBCOMPOSITE=y
CONFIG_USB_ROLE_SWITCH=y
CONFIG_USB_F_FS=y
CONFIG_USB_CONFIGFS_FS=y
```

2. 切换为 Device 模式
   - 参考本章 **切换 switch GPIO** 小节。
   - 将 OTG 控制器切到 Device 模式:

```
echo device > /proc/cviusb/otg_role
```

3. 运行脚本 run_usb.sh 并启动 adbd 脚本位于板子 /etc 目录下。adbd 需从 SDK 目录 ramdisk/rootfs/public/adbd/<TOOLCHAIN>/usr/bin/adbd 拷贝到板子后使用:

```
/etc/run_usb.sh probe adb
/etc/run_usb.sh start
./adbd &
```

4. 连接 USB 线通过 USB 将平台与 Host 相连，Host 即可识别为 USB ADB 设备。
5. 在 Windows 上使用 ADB

   （1）安装 / 获取 ADB

   · 官方 SDK Platform-Tools(推荐)打开 https://developer.android.com/studio/releases/platform-tools ，下载 Windows 版 “SDK Platform-Tools” 并解压（例如解压到 C:\platform-tools)。

   （2）把 adb 加入系统 PATH

   · 右键此电脑 -> 属性 -> 高级系统设置 -> 环境变量。

   · 在 “系统变量” 中选中 Path → “编辑” → “新建”，填入 adb 所在目录（如 C:\platform-tools)，确定保存。

   · 关闭并重新打开一个 CMD 窗口使 PATH 生效。

   （3）验证并连接板子

   在新开的 CMD 中执行:

```
adb version
```

   能输出版本信息说明 PATH 已生效。再执行:

```
adb devices
```

   若列表中出现设备（如 xxxxxxxx device)，再执行:

```
adb shell
```

   即可进入 adb 控制台。

3.4.6 Device 作为 RNDIS 设备的操作指南

1. 内核配置使能 USB Device 及 RNDIS 相关配置，例如:

```
CONFIG_USB_SUPPORT=y
CONFIG_USB=y
CONFIG_USB_DWC2=y
CONFIG_USB_GADGET=y
CONFIG_USB_CONFIGFS=y
CONFIG_USB_LIBCOMPOSITE=y
CONFIG_USB_ROLE_SWITCH=y
CONFIG_USB_CONFIGFS_RNDIS=y
CONFIG_USB_F_RNDIS=y
CONFIG_USB_ETH_RNDIS=y
CONFIG_USB_U_ETHER=y
```

2. 切换为 Device 模式

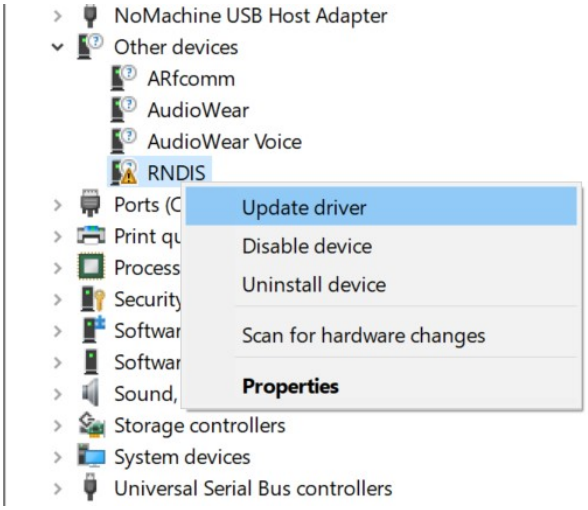
- 参考本章 **切换 switch GPIO** 小节。
- 将 OTG 控制器切到 Device 模式:

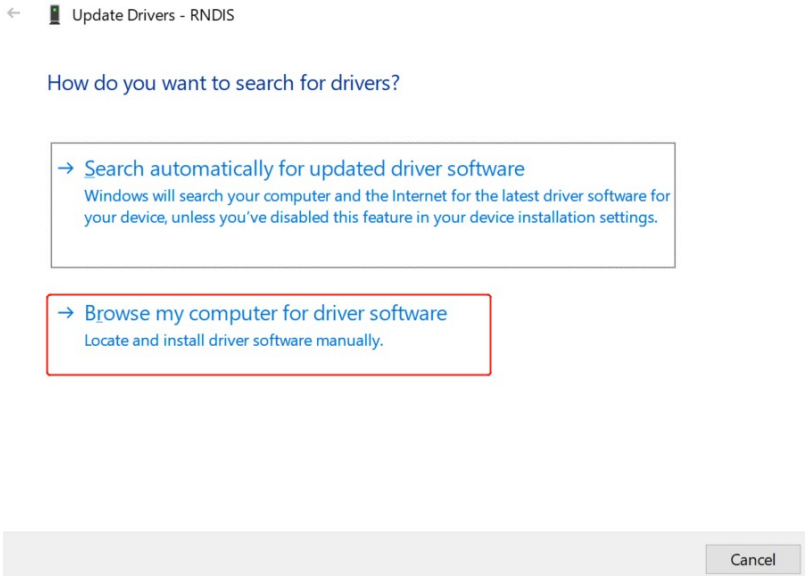
```
echo device > /proc/cviusb/otg_role
```

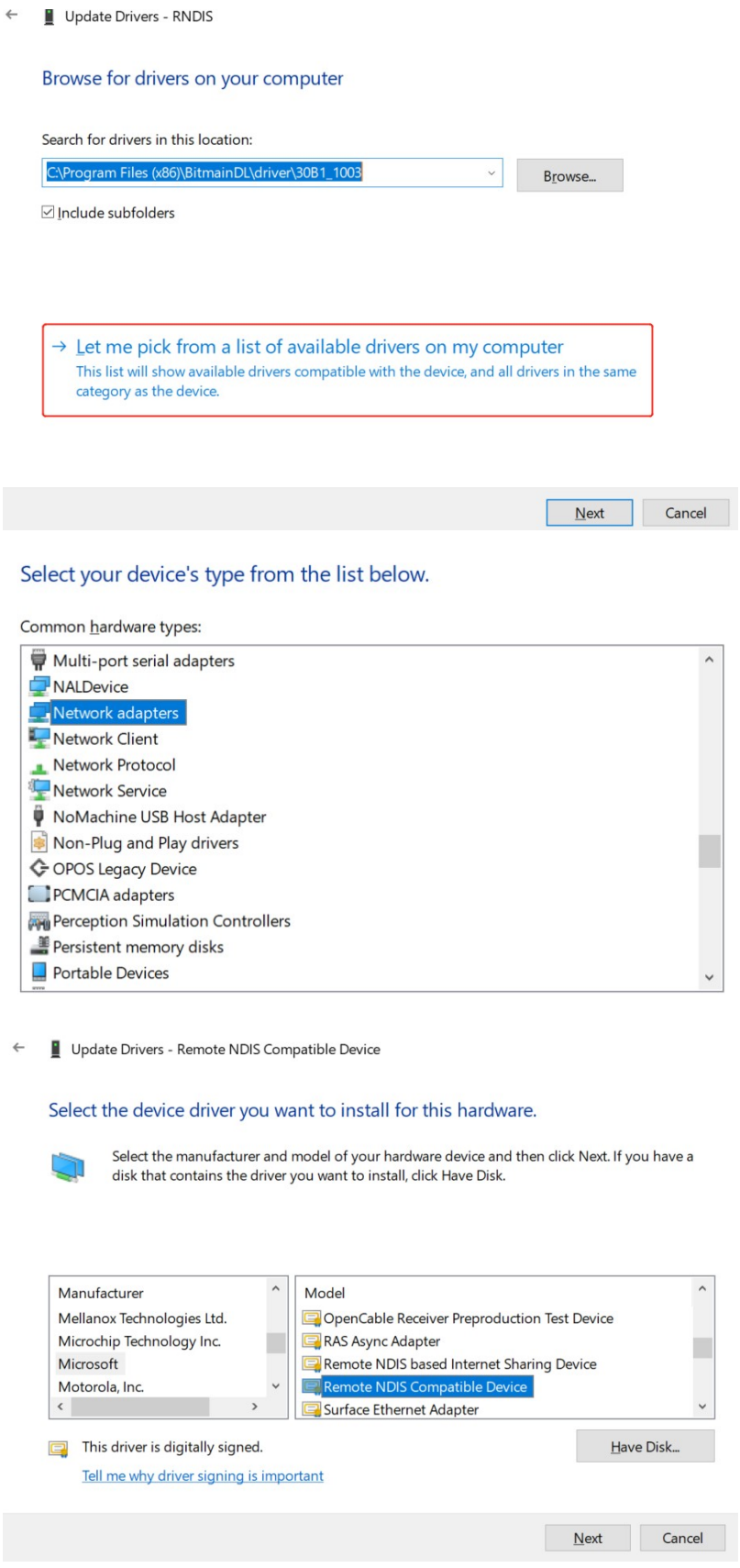
3. 运行脚本 run_usb.sh 并设置板端 IP 脚本位于板子 /etc 目录下:

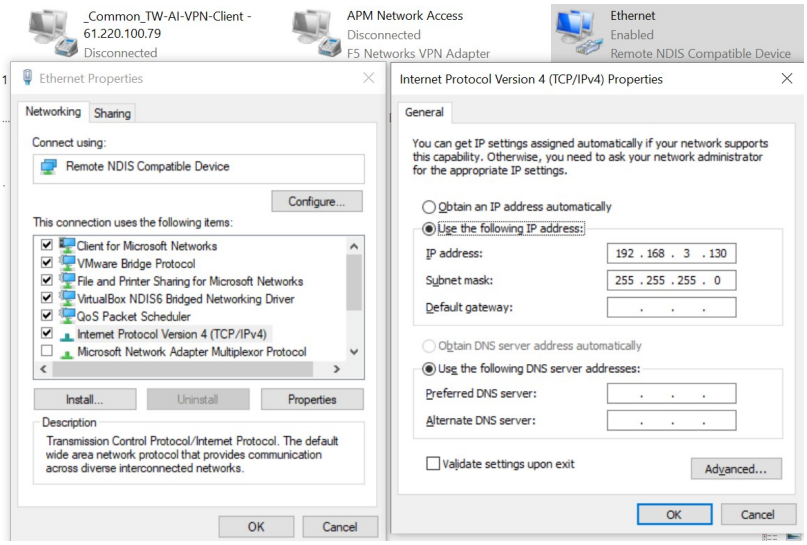
```
/etc/run_usb.sh probe rndis
/etc/run_usb.sh start
ifconfig usb0 192.168.3.101 up
```

4. 连接 USB 线通过 USB 将平台与 Host 相连，Host 即可识别为 USB Remote NDIS 设备。在 Windows 上需安装 “Remote NDIS Compatible Device” 驱动。
5. 配置 Windows 驱动在设备管理器中为 RNDIS 设备选择驱动：Device Manager -> 右键 RNDIS，后续步骤按照下图操作。板端与 Windows 端配置同网段 IP，随后互 ping 验证连通性。






Update Drivers - RNDIS
Browse for drivers on your computer
Search for drivers in this location:
C:\Program Files (x86)\BitmainDL\driver\30B1_1003
Browse...
Include subfolders
Let me pick from a list of available drivers on my computer
This list will show available drivers compatible with the device, and all drivers in the same category as the device.
Next
Cancel
Select your device's type from the list below.
Common hardware types:
Multi-port serial adapters
NALDevice
Network adapters
Network Client
Network Protocol
Network Service
NoMachine USB Host Adapter
Non-Plug and Play drivers
OPOS Legacy Device
PCMCIA adapters
Perception Simulation Controllers
Persistent memory disks
Portable Devices
Update Drivers - Remote NDIS Compatible Device
Select the device driver you want to install for this hardware.
Select the manufacturer and model of your hardware device and then click Next. If you have a disk that contains the driver you want to install, click Have Disk.
Manufacturer
Mellanox Technologies Ltd.
Microchip Technology Inc.
Microsoft
Motorola, Inc.
Model
OpenCable Receiver Preproduction Test Device
RAS Async Adapter
Remote NDIS based Internet Sharing Device
Remote NDIS Compatible Device
Surface Ethernet Adapter
This driver is digitally signed.
Tell me why driver signing is important
Have Disk...
Next
Cancel



若出现 Windows 能 ping 通板端、板端 ping 不通 Windows 的情况，可在 Windows 侧添加入站规则允许 ICMPv4（推荐做法如下）：

- 按 Win + R，输入 wf.msc，回车，打开“高级安全 Windows 防火墙”。
- 左侧点击“入站规则”，右侧点击“新建规则”。
- 选择“自定义” → 下一步。
- “此规则应用于哪些协议和端口”：协议类型选“ICMPv4”（在“自定义”或下拉中选择 ICMPv4）→ 下一步。
- 操作：选择“允许连接” → 下一步。
- 配置文件：域、专用、公用三项均勾选 → 下一步。
- 名称：填写“允许 Ping” → 完成。

完成后在板端执行（将 192.168.3.130 替换为实际 Windows IP）：

```
ping -c 4 192.168.3.130
```

重新连接 USB RNDIS 时

- 拔掉 USB 线。
- 执行 /etc/run_usb.sh stop。
- 执行 /etc/run_usb.sh probe rndis、/etc/run_usb.sh start 后再插上 USB 线。
- 板端再次配置 IP，例如 ifconfig usb0 192.168.3.101 up。

3.5 操作中需注意问题

操作中需要注意的问题如下:

- 系统开机后默认是 Host mode. 若要使用 Device mode 须加载模块并执行 USB ConfigFS 脚本. 当切换 Device 前，用户须确认以下事项:
  - USB Cable 未连接 Host。
  - 平台上的硬件须切换到对应的 USB mode。例如: 在切换到 Device mode 前，须关闭平台上的 USB 5V 供电。若平台上有带 Hub, 需关闭 Hub 电源并切换路径 Switch 到 Device mode connector。
- 切换为 Device mode 后，若要再使用 Host mode，用户须重新启动平台。
- 当平台做终端设备时，因 TTY 终端特性，若在短时间内传送大量数据, 可能造成数据遗失。用户使用此功能须注意此限制。
- 在 Uboot 下使用 USB Host 读取 U 盘时，注意若平台上有 Hub，须打开 Hub 电源并切换路径 Switch 到正确的 Connector。

4 SD/MMC 卡操作指南

4.1 操作准备

1. 使用 SDK 发布的 U-boot 和 kernel。
2. 文件系统：
   对于 SD/MMC 卡来说，SDK 仅支持 FAT 文件系统，支持可读可写。
   启动至 kernel 后需挂载至/mnt/sd 目录或根据项目需求的目录即可。
3. 可透过 fdisk 工具实现分区的工作。
4. CV184x SD 支持 2.0 与 3.0：

目前 CV184x SD/MMC 卡仅支持 3.3V VDDIO，使用者需注意。

4.2 操作过程

- 默认 SD/MMC 相关驱动模块已全部编入内核，不需再额外执行加载命令。
- 插入卡片上电启动，可以在 U-Boot 下，通过 fat 相关指令查看卡片内容。启动平台至 kernel 后，会自动扫描 SD 卡识别产生相应节点：/dev/mmcblk0 和 /dev/mmcblk0p1。
- U-Boot 下卡片不支持热插拔操作，kernel 下支持热插拔。在 kernel 下插入 SD 卡，就可以对 SD 卡进行相关的操作。具体操作请参见下文”操作示例”。

4.3 操作示例

4.3.1 U-Boot 下操作 SD 卡

SD/MMC 在 U-Boot 下仅支持 FAT 文件系统，且不支持热插拔，须在启动 U-Boot 前插入 SD 卡。

步骤 1：扫描 SD/MMC 总线

```
mmc rescan
```

步骤 2：确认 SD 卡是否被正确识别

mmc rescan 成功后，需逐一确认设备是否存在以及分区表是否正确，否则后续 fatls 会报 Couldn't find partition mmc 0:1。

a. 查看已识别的 MMC 设备列表

```
mmc list
```

输出示例：

```
mmc@04300000: 0 (SD)
mmc@04310000: 1
```

确认 SD 卡所在设备号（上例中设备 0 为 SD 卡）。

b. 设置当前设备并查看分区表

```
mmc dev 0          # 切换到 SD 卡设备（设备号以 mmc list 输出为准）
mmc info           # 查看设备信息（容量、速率等）
mmc part           # 列出分区表，确认分区号是否存在
```

输出示例：

```
Partition Map for MMC device 0  --   Partition Type: DOS

Part    Start Sector    Num Sectors     UUID            Type
  1     2048            61407232        00000000-01     0c Boot
```

上例中分区号为 1（Part 列），类型 0c 表示 FAT32（LBA）。如果 mmc part 无输出，表示 SD 卡尚未分区，需要先在 Linux 或 Windows 下分区并格式化为 FAT32。

c. 根据分区表中的实际分区号确认操作对象

如果 mmc part 输出显示分区号为 1，则后续命令使用 mmc 0:1；若为其他编号（如 2），则需相应替换为 mmc 0:2。

步骤 3：查看 SD 卡内的文件（FAT）

```
fatls mmc 0:1 /
```

其中 0 为 mmc list 确认的设备号，1 为 mmc part 确认的分区号。

步骤 4：将文件读入内存

```
fatload mmc 0:1 0x82000000 filename.bin
```

步骤 5：从内存写入文件到 SD 卡

```
fatwrite mmc 0:1 0x82000000 newfile.bin 0x100000
```

其中 0x82000000 为加载地址，0x100000 为写入字节数，请按实际环境修改。

4.3.2 Linux 下操作 SD 卡

步骤 1：检查分区

插入 SD 卡后，检查是否产生设备节点：

```
ls /dev/mmcblk0*
```

- 若显示 /dev/mmcblk0 但没有对应分区（如 mmcblk0p1），表示 SD 卡尚未分区或格式化。
- 若有 /dev/mmcblk0p1，表示已分区，可跳过步骤 2 直接挂载。

步骤 2：分区与格式化（若 SD 卡尚未分区）

```
# 创建分区（按提示操作）
fdisk /dev/mmcblk0

# 格式化为 FAT32
mkdosfs -F 32 /dev/mmcblk0p1
```

步骤 3：挂载

```
mount /dev/mmcblk0p1 /mnt/sd
```

步骤 4：读写文件

```
# 查看 SD 卡内容
ls /mnt/sd

# 从 SD 卡复制文件到本地（读）
cp /mnt/sd/data.bin ./

# 从本地复制文件到 SD 卡（写）
cp ./data.bin /mnt/sd/

# 同步写入并卸载
sync
umount /mnt/sd
```

4.4 操作中需要注意的问题

1. 需确保 SD 卡与卡槽硬件脚位接触良好，如若接触不良，有可能会出现检测错误或读写数据错误相关错误信息，并导致读写失败。
2. 每次插入 SD 卡后，都需要做一次挂载操作，才能读写 SD 卡；如果 SD 卡已经挂载到文件系统，拔卡前则必须做一次卸载（umount）操作，否则有可能在下次插入 SD 卡后看不到 SD 卡分区。另，异常拔卡亦需要进行卸载动作。
3. 必须确保 SD 卡已经创建分区，并将该分区格式化为 FAT 或 FAT32 文件系统（LINUX 下通过 fdisk 命令，Windows 下使用磁盘管理工具）。
4. 在正常操作过程中不能进行的操作：

- 读写 SD 卡时不要拔卡，否则会打印一些异常信息，并且可能会导致卡中文件或文件系统被破坏。
- 若当前目录是处于挂载目录之下如/mnt/sd 时，则无法进行卸载操作，必须离开当前目录如/mnt/sd，才能进行卸载操作。
- 系统中读写挂载目录的进程没有完全结束前，不能进行卸载操作，必须完全结束操作挂载目录的任务才能正常卸载。
- 在操作过程中出现异常时的操作：
  1. 如果因为读写数据或其它不明原因导致文件系统被破坏，读写 SD 卡时可能会出现文件系统错误信息，这时需要进行卸载操作，拔卡，再次插卡并挂载，才能再次正常读写 SD 卡。
  2. 因为 SD 卡的注册，检测/注销过程需要一定的时间，因此拔卡后若再快速地插入卡，有可能会出现检测不到 SD 卡的现象。
  3. 如果在测试过程中异常拔卡，使用者需要按 ctrl+c 以回退出到 kernel shell 下，否则会一直不停地打印异常信息。
  4. SD 卡上有一个以上的分区时，可以通过挂载操作切换挂载不同的分区，但最后需确认挂载操作的次数与卸载操作次数相等，才能确保完全卸载所有的挂载分区。

5 I2C 操作指南

本文档说明在 CV184x 平台上使用 I2C 总线进行读写操作、速率配置、GPIO 模拟 I2C、Recovery 功能以及内核态/用户态编程的完整流程。

5.1 操作准备

使用 I2C 前需满足以下条件：

· 使用官方发布的 SDK 编译后的内核镜像（默认已包含 I2C 相关驱动）。

配置文件路径一般为 build/boards/cv184x/板名/linux/板名 _defconfig 。

```
CONFIG_I2C_CHARDEV=y
CONFIG_I2C_DESIGNWARE_PLATFORM=y
```

5.1.1 切换引脚（将引脚复用到 I2C 功能）

若 I2C 使用的 SCL/SDA 引脚当前被复用作其他功能，需先切换到 I2C 功能。以下以 I2C2 为例。

1. 查看 I2C2_SCL 引脚当前功能

```
cvi_pinmux -r IIC2_SCL
# 输出显示当前功能，例如：PWR_GPIO_12（请以实际输出为准）
```

2. 将引脚功能设置为 IIC2_SCL

```
cvi_pinmux -w IIC2_SCL/IIC2_SCL
# 表示将引脚功能设置为 IIC2_SCL
```

3. 修改 I2C2_SDA 引脚

按同样方式对 IIC2_SDA 执行 cvi_pinmux -r IIC2_SDA 查看后，再使用 cvi_pinmux -w IIC2_SDA/IIC2_SDA 切换到 I2C 功能。

5.2 操作过程

1. **启动系统**默认 I2C 相关模块已编入内核，无需单独加载模块；系统启动后即可使用 I2C。
2. **进行 I2C 读写**在串口或 SSH 终端中执行下文中的 i2c-tools 命令，或在内核态/用户态编写程序，对挂载在对应 I2C 总线上的从设备进行读写。

5.3 接口速率设置说明

I2C 总线速率由设备树中的 clock-frequency 决定，单位为 Hz。修改后需重新编译内核。

DesignWare I2C 驱动仅支持以下 4 种速率，设置其他值会导致该 I2C 控制器初始化失败：

表 5.1: 支持的 I2C 速率

模式	clock-frequency 值	说明
Standard Mode	<100000>	100 kHz，标准模式
Fast Mode	<400000>	400 kHz，快速模式（默认）
Fast Mode Plus	<1000000>	1 MHz，快速模式 +
High Speed Mode	<3400000>	3.4 MHz，高速模式（目前设备不支持）

注解:

- High Speed Mode 需要硬件 IP 支持，CV184x 目前最高支持 Fast Mode，驱动会自动降回 Fast Mode。
- 频率越高对 PCB 走线质量、上拉电阻阻值、总线电容等要求越严格，建议用示波器确认信号质量。

修改位置： 文件 build/boards/default/dts/cv184x/cv184x_base.dtsi；若使用定制板，则修改对应板级 DTS 中引用的该节点或覆盖该属性。

示例： 将 i2c0 速率设为 400kHz

```
i2c0: i2c@04000000 {
  compatible = "snps,designware-i2c";
  clocks = <&clk CV184X_CLK_I2C>;
  reg = <0x0 0x04000000 0x0 0x1000>;
  clock-frequency = <400000>;

  #size-cells = <0x0>;
  #address-cells = <0x1>;
  resets = <&rst RST_I2C0>;
  reset-names = "i2c0";
};
```

5.4 I2C 读写命令示例（i2c-tools）

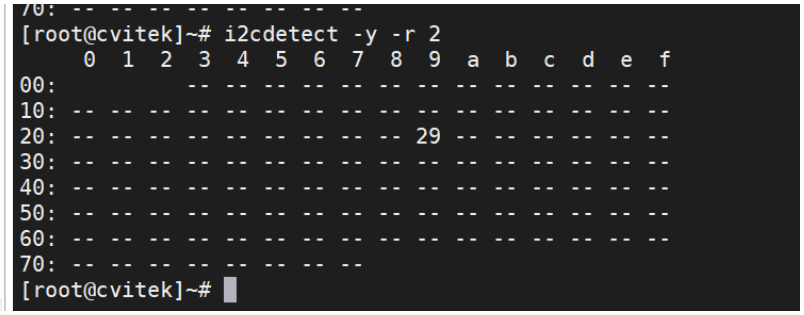
以下命令在 Linux 终端中执行，需已安装 i2c-tools。cv184x 平台常见总线编号为 i2c-0 ~ i2c-4。

1. 列出系统中的 I2C 总线

```
i2cdetect -l
```

2. 扫描某条总线上的从设备地址

```
i2cdetect -y -r N
# N 为总线编号，例如扫描 i2c-2:
i2cdetect -y -r 2
# 扫描 i2c-3:
i2cdetect -y -r 3
# 扫描 i2c-4:
i2cdetect -y -r 4
# 扫描 i2c-5:
i2cdetect -y -r 5
```



3. 批量读取从设备寄存器（以 8 位寄存器地址为例）

```
i2cdump -f -y N M
# N：总线编号；M：从设备 7 位地址（十六进制，可省略 0x 前缀）
# 示例：读取 i2c-2 上地址 0x50 的设备的全部寄存器
i2cdump -f -y 2 0x50
```

4. 读取单个寄存器

```
i2cget -f -y 0 0x3c 0x00
# 从 i2c-0 上地址 0x3c 的设备读取寄存器 0x00 的值
```

5. 写入单个寄存器

```
i2cset -f -y 0 0x3c 0x40 0x12
# 向 i2c-0 上地址 0x3c 的设备的寄存器 0x40 写入数据 0x12
```

5.5 GPIO 模拟 I2C（以 I2C-2 为例）

当硬件未引出硬件 I2C2 或需用普通 GPIO 模拟 I2C 时，可使用内核的 i2c-gpio 驱动。以下示例将 I2C 总线编号 2 配置为 GPIO 模拟。

步骤 1：修改设备树

在板级内核配置文件中确保以下选项已开启。配置文件路径一般为 build/boards/cv184x/板名/linux/板名 _defconfig 。

```
CONFIG_I2C_GPIO=y
```

编辑板级 DTS 文件，路径一般为 build/boards/cv184x/板名/dts_arm/板名.dts ，其中”板名”替换为实际板卡名称，具体以 SDK 为准。

- 关闭硬件 I2C2 节点，并添加 GPIO 模拟 I2C 节点，示例如下。可根据开发板引脚实际情况选用任意可用 GPIO 作为 SCL/SDA 进行模拟 I2C（需保证该引脚在 pinmux 中可配置为 GPIO 且未被其他外设占用）。

```
/dts-v1/;
#include "cv184x_base_arm.dtsi"
#include "cv184x_asic_bga.dtsi"
#include "cv184x_asic_spinor.dtsi"
#include "cv184x_default_memmap.dtsi"

&i2c2 {
  status = "disabled";
  scl-gpios = <0>;
  sda-gpios = <0>;
};

/ {
  sysdma_remap {
    ch-remap = <CVI_I2S0_RX CVI_I2S2_TX CVI_SPI1_RX CVI_SPI1_TX
        CVI_SPI_NOR_RX CVI_SPI_NOR_TX CVI_I2S2_RX CVI_I2S3_TX>;
  };

  aliases {
    i2c2 = &i2c2_gpio;
  };

  i2c2_gpio: i2c@2 {
    compatible = "i2c-gpio";
    i2c-gpio,delay-us = <25>;   /* 约 20kHz，低速更稳定 */
    gpios = <&porte 13 (GPIO_ACTIVE_HIGH | GPIO_OPEN_DRAIN)>,
          <&porte 12 (GPIO_ACTIVE_HIGH | GPIO_OPEN_DRAIN)>;
    i2c-gpio,sda-open-drain;
    i2c-gpio,scl-open-drain;
    #address-cells = <1>;
    #size-cells = <0>;
    status = "okay";
    linux,init-delay = <200>;  /* 上电延迟 200ms，避免与其它外设冲突 */
  };
};
```

说明： GPIO 引脚（如 porte 12/13）需根据实际原理图修改为当前板子使用的 SDA/SCL 引脚。其中 porte 可根据 GPIO 所在的组改为 porta、portb、portc、portd 或 porte，引脚号为该组内 0 ~ 31。若不需要固定为 i2c-2，可注释掉 aliases 部分，由内核自动分配总线号，从而实现绑定到其他总线（如 i2c-4、i2c-5 等）。

步骤 2：验证 GPIO 模拟 I2C-2

重新编译并烧录内核后，在终端执行：

1. 列出 I2C 总线，确认存在 i2c-2：

```
i2cdetect -l
```

```
[root@cvitek]/mnt/sd# i2cdetect -l
i2c-3   i2c             Synopsys DesignWare I2C adapter         I2C adapter
i2c-4   i2c             Synopsys DesignWare I2C adapter         I2C adapter
i2c-2   i2c             i2c@2                                   I2C adapter
i2c-0   i2c             Synopsys DesignWare I2C adapter         I2C adapter
i2c-5   i2c             Synopsys DesignWare I2C adapter         I2C adapter
```

2. 扫描 i2c-2 上的设备：

```
i2cdetect -y 2
```

```
[root@cvitek]~# i2cdetect -y 2
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: 50 51 52 53 54 55 56 57 -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

3. 写入并读回数据（示例地址 0x50）：

```
i2cset -y 2 0x50 0x00 0xAA
i2cget -y 2 0x50 0x00
# 正常时应输出 0xaa
```

5.6 I2C Recovery 配置说明

当 I2C 总线发生死锁（如从设备拉低 SDA 不释放）时，可通过 I2C Recovery 功能由内核在检测到超时后切换为 GPIO 模式模拟时钟，尝试恢复总线。该功能在默认设备树中可能被注释或未使能，需要手动在板级 DTS 中配置。

启用步骤：

1. **编辑板级设备树**打开：build/boards/cv184x/board_name/dts_arm/board_name.dts （将 board_name 替换为实际板卡名称）。

2. **为对应 I2C 节点添加 Recovery 相关属性**在需要 Recovery 的 &i2cN 节点中增加 scl-pinmux、sda-pinmux、scl-gpios、sda-gpios，使内核在 Recovery 时能通过 GPIO 驱动 SCL/SDA。
3. **重新编译内核**保存 DTS 后重新编译内核/设备树并烧录，使配置生效。

各 I2C 控制器 Recovery 配置示例（仅供参考，请根据板子实际使用的 SCL/SDA 引脚修改）：

I2C0：

```
&i2c0 {
    scl-pinmux = <0x03001070 0x0 0x3>;    /* IIC0_SCL → XGPIOA[28] */
    sda-pinmux = <0x03001074 0x0 0x3>;    /* IIC0_SDA → XGPIOA[29] */
    scl-gpios = <&porta 28 GPIO_ACTIVE_HIGH>;
    sda-gpios = <&porta 29 GPIO_ACTIVE_HIGH>;
};
```

I2C1：

```
&i2c1 {
    scl-pinmux = <0x0300111c 0x2 0x3>;    /* IIC1_SCL → XGPIOB[7] */
    sda-pinmux = <0x03001118 0x2 0x3>;    /* IIC1_SDA → XGPIOB[8] */
    scl-gpios = <&portb 7 GPIO_ACTIVE_HIGH>;
    sda-gpios = <&portb 8 GPIO_ACTIVE_HIGH>;
};
```

I2C2：

```
&i2c2 {
    scl-pinmux = <0x030010b8 0x0 0x3>;    /* IIC2_SCL → XPWR_GPIO[12] */
    sda-pinmux = <0x030011bc 0x0 0x3>;    /* IIC2_SDA → XPWR_GPIO[13] */
    scl-gpios = <&porte 12 GPIO_ACTIVE_HIGH>;
    sda-gpios = <&porte 13 GPIO_ACTIVE_HIGH>;
};
```

I2C3：

```
&i2c3 {
    scl-pinmux = <0x03001014 0x0 0x3>;    /* IIC3_SCL → XGPIOA[5] */
    sda-pinmux = <0x03001018 0x0 0x3>;    /* IIC3_SDA → XGPIOA[6] */
    scl-gpios = <&porta 5 GPIO_ACTIVE_HIGH>;
    sda-gpios = <&porta 6 GPIO_ACTIVE_HIGH>;
};
```

I2C4：

```
&i2c4 {
    scl-pinmux = <0x030010f0 0x2 0x3>;    /* IIC4_SCL → XGPIOB[1] */
    sda-pinmux = <0x030010f4 0x2 0x3>;    /* IIC4_SDA → XGPIOB[2] */
    scl-gpios = <&portb 1 GPIO_ACTIVE_HIGH>;
    sda-gpios = <&portb 2 GPIO_ACTIVE_HIGH>;
};
```

参数说明：

- scl-pinmux 和 sda-pinmux: 定义 SCL 和 SDA 引脚的复用配置
  - 第一个参数：引脚复用寄存器地址

  - 第二个参数：I2C 功能选择值
  - 第三个参数：GPIO 功能选择值
- scl-gpios 和 sda-gpios: 定义 recovery 时使用的 GPIO 引脚
  - 格式：<&gpio_port pin_number polarity>
  - GPIO_ACTIVE_HIGH 表示高电平有效

注解:

- 启用 I2C recovery 功能后，当检测到总线死锁时，系统会自动切换到 GPIO 模式进行总线恢复
- 只有在遇到 I2C 总线死锁问题时才需要启用此功能
- 修改设备树后需要重新编译内核才能生效

5.7 内核态 I2C 读写程序示例

以下展示在内核驱动中通过 I2C 子系统访问从设备的标准流程与完整示例代码。

流程概述：

1. 使用 i2c_get_adapter(bus_id) 获取 I2C 适配器。
2. 使用 i2c_new_client_device() 在总线上创建设备客户端（5.10 内核推荐，失败时返回 ERR_PTR；亦可使用 i2c_new_probed_device 等）。
3. 使用 i2c_master_send / i2c_master_recv 进行读写；使用完毕后 i2c_unregister_device、i2c_put_adapter。

完整内核模块示例（含头文件、初始化、写寄存器、读寄存器、模块入口）：

```c
#include <linux/module.h>
#include <linux/i2c.h>
#include <linux/err.h>
#include <linux/errno.h>
#include <linux/kernel.h>

/* 设备地址需与实际从设备一致；总线号对应 i2c-0, i2c-1, ... */
#define I2C_DEV_ADDR    0x3c
#define I2C_BUS_NUM     0

static struct i2c_client *client;

/* 写寄存器：buf[0]=寄存器地址，buf[1]=数据 */
static int __used i2c_dev_write(u8 reg, u8 value)
{
    u8 buf[2] = { reg, value };
    int ret = i2c_master_send(client, buf, sizeof(buf));
    if (ret < 0)
        pr_err("Write failed: %d\n", ret);
```

（下页继续）

(续上页)

```
    return ret;
}

/* 读寄存器：先发寄存器地址，再读 1 字节 */
static int __used i2c_dev_read(u8 reg, u8 *out_val)
{
    int ret;

    ret = i2c_master_send(client, &reg, 1);
    if (ret < 0) {
        pr_err("Register select failed: %d\n", ret);
        return ret;
    }

    ret = i2c_master_recv(client, out_val, 1);
    if (ret < 0)
        pr_err("Read failed: %d\n", ret);
    return ret;
}

static int __init cvi_i2c_dev_init(void)
{
    struct i2c_adapter *adapter;
    struct i2c_board_info info = {
        I2C_BOARD_INFO("dummy", I2C_DEV_ADDR),
    };

    adapter = i2c_get_adapter(I2C_BUS_NUM);
    if (!adapter) {
        pr_err("I2C adapter not found\n");
        return -ENODEV;
    }

    /* 5.10 内核使用 i2c_new_client_device，失败时返回 ERR_PTR */
    client = i2c_new_client_device(adapter, &info);
    i2c_put_adapter(adapter);

    if (IS_ERR(client)) {
        pr_err("Device registration failed: %ld\n", PTR_ERR(client));
        return PTR_ERR(client);
    }

    pr_info("I2C device initialized\n");
    return 0;
}

static void __exit cvi_i2c_dev_exit(void)
{
    i2c_unregister_device(client);
    pr_info("I2C device removed\n");
}

module_init(cvi_i2c_dev_init);
module_exit(cvi_i2c_dev_exit);
```

(下页继续)

(续上页)

```

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("CVI I2C device driver - kernel I2C read/write example");
MODULE_AUTHOR("Your Name");
```

5.8 用户态 I2C 读写程序示例

以下演示在用户空间通过 /dev/i2c-N 与 ioctl(I2C_SLAVE / I2C_RDWR) 访问 I2C 设备。

注解:

- 内核需开启 CONFIG_I2C_CHARDEV，并确保当前用户有权限访问 /dev/i2c-*。

完整用户态程序示例（含头文件、写寄存器、读寄存器及 I2C_M_RD 兼容）:

```c
/*
 * 用户态 I2C 读写示例：通过 /dev/i2c-N 与 ioctl(I2C_SLAVE / I2C_RDWR) 访问 I2C 设备。
 * 需内核开启 CONFIG_I2C_CHARDEV，且当前用户有权限访问 /dev/i2c-*。
 */
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdint.h>
#include <linux/i2c-dev.h>
#include <sys/ioctl.h>

#if defined(__has_include) && __has_include(<linux/i2c.h>)
# include <linux/i2c.h>
#else
/* 工具链无 linux/i2c.h 时提供与内核 ABI 一致的最小定义 */
# ifndef I2C_M_RD
#  define I2C_M_RD 0x0001
# endif
struct i2c_msg {
    uint16_t addr;
    uint16_t flags;
    uint16_t len;
    uint8_t *buf;
};
#endif

#define I2C_DEV_PATH    "/dev/i2c-0"
#define I2C_SLAVE_ADDR  0x3C   /* 7 位从设备地址 */

/* 写寄存器：先发寄存器地址，再发数据 */
static int i2c_write_reg(int fd, uint8_t reg, uint8_t value)
{
    uint8_t buf[2] = { reg, value };

```

(下页继续)

(续上页)

```c
    if (write(fd, buf, sizeof(buf)) != (ssize_t)sizeof(buf)) {
        perror("Write failed");
        return -1;
    }
    return 0;
}

/* 读寄存器：先写寄存器地址，再读数据（使用 I2C_RDWR 组合为一笔事务） */
static int i2c_read_reg(int fd, uint8_t reg_addr, uint8_t *data)
{
    struct i2c_msg messages[2];
    struct i2c_rdwr_ioctl_data packet;

    messages[0].addr  = I2C_SLAVE_ADDR;
    messages[0].flags = 0;          /* 写 */
    messages[0].len   = 1;
    messages[0].buf   = &reg_addr;

    messages[1].addr  = I2C_SLAVE_ADDR;
    messages[1].flags = I2C_M_RD;   /* 读 */
    messages[1].len   = 1;
    messages[1].buf   = data;

    packet.msgs  = messages;
    packet.nmsgs = 2;

    if (ioctl(fd, I2C_RDWR, &packet) < 0) {
        perror("Read failed");
        return -1;
    }
    return 0;
}

int main(void)
{
    int i2c_fd;
    uint8_t reg = 0x00;
    uint8_t value = 0x55;
    uint8_t data = 0;

    /* 步骤 1：打开 I2C 总线设备并设置从设备地址 */
    i2c_fd = open(I2C_DEV_PATH, O_RDWR);
    if (i2c_fd < 0) {
        perror("Failed to open I2C device");
        return EXIT_FAILURE;
    }

    if (ioctl(i2c_fd, I2C_SLAVE, I2C_SLAVE_ADDR) < 0) {
        perror("Failed to set slave address");
        close(i2c_fd);
        return EXIT_FAILURE;
    }

    /* 步骤 2：写入寄存器 */
```

(下页继续)

(续上页)

```
90      if (i2c_write_reg(i2c_fd, reg, value) < 0) {
91          close(i2c_fd);
92          return EXIT_FAILURE;
93      }
94
95      /* 步骤 3: 读取寄存器 */
96      if (i2c_read_reg(i2c_fd, reg, &data) < 0) {
97          close(i2c_fd);
98          return EXIT_FAILURE;
99      }
100
101     printf("Read value: 0x%02X\n", data);
102
103     close(i2c_fd);
104     return EXIT_SUCCESS;
105 }
```

6 SPI 操作指南

本文档说明在 CV184x 平台上使用 SPI 总线进行读写操作、环回测试以及内核态/用户态编程的流程。

平台说明: CV184x 上的 SPI 控制器仅支持主机 (Master) 模式，不支持配置为从机 (Slave) 模式。即本平台只能作为 SPI 总线的主设备发起传输、连接 Flash/传感器等从设备，无法作为从设备被其他主控通过 SPI 访问。本平台 SPI 时钟最高支持 46.8MHz，设备树与驱动中 spi-max-frequency 仅可配置至 46800000（Hz），超出该值可能工作异常或不支持。

6.1 操作准备

使用 SPI 前需满足以下条件:

- 使用官方发布的 SDK 编译后的内核镜像（默认已包含 SPI 相关驱动）。

6.1.1 使能内核 SPI 与 DMA 配置

本步骤分为三部分：在板级内核配置中开启 SPI 与 DMA 相关选项；在设备树中为 SPI1 启用 DMA；如需使用 DMA 通道重映射，则在板级 DTS 根节点下配置 sysdma_remap。路径中的”板名”请替换为实际使用的板级目录名（如 cv1842hp_wevb_0014a_spinor），具体以 SDK 为准。

（1）内核与 DMA 相关配置

在板级内核配置文件中确保以下选项已开启。配置文件路径一般为 build/boards/cv184x/板名/linux/板名 _defconfig 。

```
CONFIG_DMADEVICES=y
CONFIG_DW_DMAC_CVITEK=y
CONFIG_SPI=y
CONFIG_SPI_MASTER=y
CONFIG_SPI_SPIDEV=y
CONFIG_SPI_DESIGNWARE=y
CONFIG_SPI_DW_DMA=y
CONFIG_SPI_DW_MMIO=y
```

若不需要 DMA 传输（例如调试、对比性能或外设不兼容 DMA），可将 CONFIG_SPI_DW_DMA 注释掉，使 SPI 控制器回退到 PIO（中断/轮询）模式:

```
CONFIG_DMADEVICES=y
CONFIG_DW_DMAC_CVITEK=y
CONFIG_SPI=y
CONFIG_SPI_MASTER=y
CONFIG_SPI_SPIDEV=y
CONFIG_SPI_DESIGNWARE=y
# CONFIG_SPI_DW_DMA is not set
CONFIG_SPI_DW_MMIO=y
```

同时需在设备树中将对应 SPI 节点的 #if 1 改为 #if 0，关闭 DMA 属性（见下文（2）中的示例）。修改后重新编译内核与设备树并烧录至开发板即可。

（2）在设备树中为 SPI1 启用 DMA

在板级 DTS 或其所包含的 DTSI 中，找到 SPI1 节点（如 spi1@04190000），将其中的 #if 预编译开关置为 1，使 dmas、dma-names、capability 等 DMA 相关属性被编译进设备树，从而开启 SPI1 的 DMA 功能。若 #if 为 0 或未定义该块，则 SPI1 使用 PIO 模式。

```
spi1: spi1@04190000 {
  compatible = "snps,dw-apb-ssi";
  reg = <0x0 0x04190000 0x0 0x10000>;
  clocks = <&clk CV184X_CLK_APB_SPI1>;
  #address-cells = <1>;
  #size-cells = <0>;
  #if 1
  dmas = <&dmac 2 1 1
        &dmac 3 1 1>;
  dma-names = "rx", "tx";
  capability = "txrx";
  #endif
  status = "okay";
  num-cs = <1>;
  spidev@0 {
    compatible = "rohm,dh2228fv";
    spi-max-frequency = <1000000>;  /* 最高支持 46.8MHz，可设为 46800000 */
    reg = <0>;
  };
};
```

spi-max-frequency 单位为 Hz，本平台 SPI IP 仅支持最高 46.8MHz，请勿配置超过 46800000。

（3）DMA 通道重映射

若板级需要指定 DMA 通道与外设的对应关系，可在板级 DTS 的根节点 / 下添加或合并 sysdma_remap 节点。板级 DTS 路径一般为 build/boards/cv184x/板名/dts_arm/板名.dts 。下例中 CVI_SPI1_RX、CVI_SPI1_TX 与 SPI1 设备节点中的 DMA 通道 2 和 3 对应，其余通道按实际需求配置。

```
/ {
  sysdma_remap {
    ch-remap = <CVI_I2S0_RX CVI_I2S2_TX CVI_SPI1_RX CVI_SPI1_TX
          CVI_SPI_NOR_RX CVI_SPI_NOR_TX CVI_I2S2_RX CVI_I2S3_TX>;
  };
};
```

6.1.2 在设备树中将 SPI1 使能

在板级 DTS 中将 SPI1 节点设为 status = "okay"，其余不使用的 SPI 节点设为 status = "disabled"。

6.1.3 SPI1 引脚切换

说明：以下”需切换两次引脚功能”仅为 **SPI1** 的特殊情况（先切到 MUX 再指到具体功能）。其他 SPI 控制器或外设的 Pinmux 请根据实际板级原理图与引脚定义操作，可能只需一次切换或不同顺序，以 pinout 文档为准。

步骤 1. 将 PAD_MIPIRX3P 切 MUX_SPI1_MOSI

```
cvi_pinmux -w PAD_MIPIRX3P/MUX_SPI1_MOSI
```

步骤 2. 将 MUX_SPI1_MOSI 切 SPI1_SDO

```
cvi_pinmux -w MUX_SPI1_MOSI/SPI1_SDO
```

步骤 3. MISO 引脚

按同样方式进行切换（先切换到 MUX_SPI1_MISO，再切到 SPI1_SDI ，具体以板级引脚定义为准）。

```
cvi_pinmux -w PAD_MIPIRX3N/MUX_SPI1_MISO

cvi_pinmux -w MUX_SPI1_MISO/SPI1_SDI
```

6.2 操作过程

1. **启动系统** SPI 相关驱动已编入内核，无需单独加载模块；启动后即可使用 SPI。
2. **进行 SPI 读写**在串口或 SSH 终端中运行 SPI 读写命令，或在内核态/用户态编写程序，对挂载在 SPI 控制器上的从设备进行读写。

6.3 SPI 环回测试

环回测试用于验证 SPI 控制器与引脚是否工作正常：将 SPI1 的 MOSI 与 MISO 短接，发送数据后读回，比对是否一致。

步骤 1：短接 SPI1 的 MOSI 与 MISO 引脚

步骤 2：完成 SPI1 内核配置与设备树使能

按上文「操作准备」中的「使能内核 SPI 与 DMA 配置」和「在设备树中将 SPI1 使能」两节完成配置，重新编译内核与设备树并烧录至开发板。

步骤 3：确认 spidev 节点

```
ls /dev/spidev*
# 示例输出：/dev/spidev1.0，后续命令中的设备节点需与输出一致
```

步骤 4：执行 cvi_pinmux 切换

以下「需切换两次引脚功能」仅为 **SPI1** 的特殊情况（先切到 MUX 再指到具体功能）。其他 SPI 控制器或外设的 Pinmux 请根据实际板级原理图与引脚定义操作，可能只需一次切换或不同顺序，以 pinout 文档为准。

步骤 4.1 将 PAD_MIPIRX3P 切 MUX_SPI1_MOSI

```
cvi_pinmux -w PAD_MIPIRX3P/MUX_SPI1_MOSI
```

步骤 4.2 将 MUX_SPI1_MOSI 切 SPI1_SDO

```
cvi_pinmux -w MUX_SPI1_MOSI/SPI1_SDO
```

步骤 4.3 MISO 引脚

按同样方式进行切换（先切换到 MUX_SPI1_MISO，再切到 SPI1_SDI，具体以板级引脚定义为准）。

```
cvi_pinmux -w PAD_MIPIRX3N/MUX_SPI1_MISO

cvi_pinmux -w MUX_SPI1_MISO/SPI1_SDI
```

步骤 5：编译并部署 spidev_test

在 SDK 的 linux_5.10/tools/spi 目录下执行 make，将生成的 spidev_test 拷贝到板端。

步骤 6：生成测试数据

```
dd if=/dev/urandom of=data.bin bs=2k count=1
```

步骤 7：执行环回测试

```
./spidev_test -D /dev/spidev1.0 -s 10000000 -i data.bin -o data_out.bin
# -D 为设备节点，-s 为速率(Hz)，-i 为输入文件，-o 为输出文件
```

步骤 8：比对数据

```
diff data.bin data_out.bin
# 无输出表示两文件一致，环回测试通过
```

环回测试通过后，可连接实际 SPI 从设备进行读写操作。

6.4 操作示例

6.5 内核态 SPI 读写程序示例

以下说明在内核驱动中通过 SPI 子系统访问从设备的典型流程与示例代码。

流程概述：

1. 使用 spi_busnum_to_master(bus_num) 获取 SPI 控制器（spi_master），bus_num 为目标设备所在的总线号。
2. 根据总线名与片选号找到 spi_device``（如通过 ``bus_find_device_by_name 等）。
3. 填充 spi_transfer 并加入 spi_message，再调用 spi_sync 或 spi_async 执行同步或异步传输。

步骤 1：获取 SPI 控制器

```
master = spi_busnum_to_master(bus_num);
/* bus_num：设备所在 SPI 总线号；master：spi_master 指针 */
```

步骤 2：获取 SPI 从设备

```
snprintf(str, sizeof(str), "%s.%u", dev_name(&master->dev), cs);
dev = bus_find_device_by_name(&spi_bus_type, NULL, str);
spi = to_spi_device(dev);
/* cs：片选号；spi：spi_device 指针 */
```

步骤 3：组织 spi_message

```
spi_message_init(&m);
spi_message_add_tail(&t, &m);
/* t：spi_transfer；m：spi_message */
```

步骤 4：发起传输

```
status = spi_sync(spi, &m);  /* 同步 */
/* 或 status = spi_async(spi, &m);  异步 */
```

完整示例代码（仅供参考，非可直接运行的完整模块）：

```
#include <linux/spi/spi.h>

static unsigned int busnum;
module_param(busnum, uint, 0);
MODULE_PARM_DESC(busnum, "SPI bus number (default=0)");

static unsigned int cs;
module_param(cs, uint, 0);
MODULE_PARM_DESC(cs, "SPI chip select (default=0)");

extern struct bus_type spi_bus_type;
static struct spi_master *master;
static struct spi_device *spi_device;
```

（下页继续）

(续上页)

```
14
15  static int __init spidev_init(void)
16  {
17      char *spi_name;
18      struct device *dev;
19
20      master = spi_busnum_to_master(busnum);
21      if (!master)
22          return -ENODEV;
23
24      spi_name = kzalloc(strlen(dev_name(&master->dev)) + 8, GFP_KERNEL);
25      if (!spi_name)
26          return -ENOMEM;
27
28      snprintf(spi_name, strlen(dev_name(&master->dev)) + 8,
29              "%s.%u", dev_name(&master->dev), cs);
30      dev = bus_find_device_by_name(&spi_bus_type, NULL, spi_name);
31      kfree(spi_name);
32      if (!dev)
33          return -ENODEV;
34
35      spi_device = to_spi_device(dev);
36      put_device(dev);
37      if (!spi_device)
38          return -ENODEV;
39
40      return 0;
41  }
42
43  static int spi_dev_write(void *buf, unsigned long len, int buswidth)
44  {
45      struct spi_transfer t = {
46          .speed_hz = 2000000,
47          .tx_buf = buf,
48          .len = len,
49      };
50      struct spi_message m;
51
52      if (!spi_device)
53          return -ENODEV;
54
55      spi_device->mode = SPI_MODE_0;
56      t.bits_per_word = (buswidth == 16) ? 16 : 8;
57
58      spi_message_init(&m);
59      spi_message_add_tail(&t, &m);
60      return spi_sync(spi_device, &m);
61  }
62
63  static int spi_dev_read(unsigned char devaddr, unsigned char reg_addr,
64                          void *buf, size_t len)
65  {
66      u8 txbuf[4] = { devaddr, 0, reg_addr, 0 };
67      struct spi_transfer t = {
```

(下页继续)

(续上页)

```
        .speed_hz = 2000000,
        .tx_buf = txbuf,
        .rx_buf = buf,
        .len = len,
    };
    struct spi_message m;

    if (!spi_device)
        return -ENODEV;

    spi_device->mode = SPI_MODE_0;
    /* txbuf 长度与位宽需根据从设备协议填写，此处为示例 */
    spi_message_init(&m);
    spi_message_add_tail(&t, &m);
    return spi_sync(spi_device, &m);
}
```

6.6 用户态 SPI 读写程序示例

以下说明在用户空间通过 spidev 设备节点与 ioctl 进行 SPI 读写的步骤。完整实现可参考 SDK 中 tools/spi/spidev_test.c。

步骤 1：打开设备节点

```
static const char *device = "/dev/spidev1.0";
int fd = open(device, O_RDWR);
if (fd < 0)
    /* 错误处理 */;
```

注解： 设备节点与 SPI 控制器对应关系（以实际 ls /dev/spidev* 为准）：

- 正常情况：SPI 控制器 0 默认为 /dev/spidev1.0，SPI 控制器 1 为 /dev/spidev2.0，以此类推（控制器 N 对应 spidev(N+1).0）。
- 若禁用了某个 SPI 控制器（如禁用控制器 0），则其后控制器节点号前移：此时 SPI 控制器 1 为 /dev/spidev1.0，控制器 2 为 /dev/spidev2.0，以此类推。

其余操作与控制器 0 相同。

步骤 2：设置 SPI 模式

```
ret = ioctl(fd, SPI_IOC_WR_MODE32, &mode);
if (ret == -1)
    /* 错误处理 */;
ret = ioctl(fd, SPI_IOC_RD_MODE32, &mode);
```

mode 取值参见内核头文件 include/linux/spi/spi.h，例如：

```
#define SPI_CPHA  0x01  /* 时钟相位 */
#define SPI_CPOL  0x02  /* 时钟极性 */
```

(下页继续)

(续上页)

```
#define SPI_MODE_0  (0|0)
#define SPI_MODE_1  (0|SPI_CPHA)
#define SPI_MODE_2  (SPI_CPOL|0)
#define SPI_MODE_3  (SPI_CPOL|SPI_CPHA)
/* 示例：mode = SPI_MODE_3 | SPI_LSB_FIRST; */
```

步骤 3：设置每字位数（bits per word）

```
ret = ioctl(fd, SPI_IOC_WR_BITS_PER_WORD, &bits);
if (ret == -1)
    /* 错误处理 */;
ret = ioctl(fd, SPI_IOC_RD_BITS_PER_WORD, &bits);
```

步骤 4：设置传输速率

```
ret = ioctl(fd, SPI_IOC_WR_MAX_SPEED_HZ, &speed);
if (ret == -1)
    /* 错误处理 */;
ret = ioctl(fd, SPI_IOC_RD_MAX_SPEED_HZ, &speed);
```

一般建议 speed 不超过 25MHz，具体以从设备规格为准。

步骤 5：发送/接收一帧数据

```
ret = ioctl(fd, SPI_IOC_MESSAGE(1), &tr);
if (ret < 1)
    /* 错误处理 */;
```

tr 为 struct spi_ioc_transfer 数组首地址，用于描述本帧传输的缓冲区与长度。

完整可运行示例

将上述步骤串联，并加上头文件、struct spi_ioc_transfer 的填充与缓冲区，即可得到可直接编译运行的程序（需内核提供 /dev/spidev* 及对应头文件）：

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <errno.h>
5  #include <fcntl.h>
6  #include <unistd.h>
7  #include <sys/ioctl.h>
8  #include <linux/spi/spidev.h>
9
10 #define SPI_DEV   "/dev/spidev1.0"
11 #define SPI_SPEED 10000000
12 #define SPI_BITS  8
13
14 int main(void)
15 {
16     int fd;
17     int ret;
18     __u8 mode = SPI_MODE_0;
19     __u8 bits = SPI_BITS;
20     __u32 speed = SPI_SPEED;
```

(下页继续)

(续上页)

```
    __u8 tx_buf[4] = { 0x01, 0x02, 0x03, 0x04 };
    __u8 rx_buf[4] = { 0 };

    struct spi_ioc_transfer tr = {
        .tx_buf = (unsigned long)tx_buf,
        .rx_buf = (unsigned long)rx_buf,
        .len = 4,
        .speed_hz = speed,
        .bits_per_word = bits,
    };

    fd = open(SPI_DEV, O_RDWR);
    if (fd < 0) {
        perror("open " SPI_DEV);
        return EXIT_FAILURE;
    }

    ret = ioctl(fd, SPI_IOC_WR_MODE, &mode);
    if (ret == -1) {
        perror("SPI_IOC_WR_MODE");
        close(fd);
        return EXIT_FAILURE;
    }

    ret = ioctl(fd, SPI_IOC_WR_BITS_PER_WORD, &bits);
    if (ret == -1) {
        perror("SPI_IOC_WR_BITS_PER_WORD");
        close(fd);
        return EXIT_FAILURE;
    }

    ret = ioctl(fd, SPI_IOC_WR_MAX_SPEED_HZ, &speed);
    if (ret == -1) {
        perror("SPI_IOC_WR_MAX_SPEED_HZ");
        close(fd);
        return EXIT_FAILURE;
    }

    ret = ioctl(fd, SPI_IOC_MESSAGE(1), &tr);
    if (ret < 1) {
        perror("SPI_IOC_MESSAGE");
        close(fd);
        return EXIT_FAILURE;
    }

    printf("TX: %02x %02x %02x %02x\n", tx_buf[0], tx_buf[1], tx_buf[2], tx_buf[3]);
    printf("RX: %02x %02x %02x %02x\n", rx_buf[0], rx_buf[1], rx_buf[2], rx_buf[3]);

    close(fd);
    return EXIT_SUCCESS;
}
```

编译与运行：arm-none-linux-uclibcgnueabihf-gcc -o spidev_demo spidev_demo.c -static，在目标板上执行 ./spidev_demo。设备节点与 SPI 控制器对应关系见上文备注；速率、模式、字长可按从设备要求修改。

7 GPIO 操作指南

7.1 操作准备

- 使用 SDK 发布的 kernel 内核

7.2 操作流程

- GPIO 相关模块默认已编入内核，无需执行加载命令
- 通过控制台执行 GPIO 读写命令，或在内核态/用户态编写 GPIO 操作程序即可

7.3 操作示例

7.3.1 GPIO 命令行操作示例

步骤 1：导出 GPIO 控制权

```
echo N > /sys/class/gpio/export
```

- N = GPIO 组号值 + 偏移值
- 示例：GPIO1_2 引脚（GPIO1 组号 448 + 偏移 2 → N=450）

GPIO 组号对应表：

GPIO 组	Linux 组号值
GPIO0 (GPIOA)	480
GPIO1 (GPIOB)	448
GPIO2 (GPIOC)	416
GPIO3 (GPIOD)	384
PWR_GPIO (GPIOE)	352

步骤 2：配置 GPIO 方向

```
# 设置为输入模式
echo in > /sys/class/gpio/gpioN/direction

# 设置为输出模式
echo out > /sys/class/gpio/gpioN/direction
```

步骤 3：GPIO 读写操作

```
# 读取输入值
cat /sys/class/gpio/gpioN/value

# 输出低电平
echo 0 > /sys/class/gpio/gpioN/value

# 输出高电平
echo 1 > /sys/class/gpio/gpioN/value
```

步骤 4：释放 GPIO 资源

```
echo N > /sys/class/gpio/unexport
```

注解： 开启 CONFIG_DEBUG_FS 选项后，可通过以下命令查看 GPIO 映射关系：

```
cat /sys/kernel/debug/gpio
```

7.3.2 内核态 GPIO 操作示例

以下为完整的内核态 GPIO 操作示例，以可加载内核模块形式实现。示例以 GPIO1_2（编号 448 + 2 = 450）为例，实际使用时请根据 GPIO 组号对应表修改 GPIO_TEST_NUM 宏定义。模块加载后依次执行 GPIO 读写测试和中断注册，卸载时自动释放资源。

7.3.2.1 头文件与宏定义

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>
#include <linux/delay.h>

/* 示例 GPIO：GPIO1_2 = 448 + 2 */
#define GPIO_TEST_NUM  450
/* 用作 request_irq / free_irq 的 dev_id 标识 */
#define GPIO_IRQ_DEV_ID ((void *)GPIO_TEST_NUM)
```

7.3.2.2 中断回调函数

GPIO 电平发生变化时由内核调用，打印当前电平用于调试。

```
static irqreturn_t gpio_irq_handler(int irq, void *dev_id)
{
    int val = gpio_get_value(GPIO_TEST_NUM);
    printk(KERN_INFO "GPIO%d IRQ triggered, current value: %d\n",
           GPIO_TEST_NUM, val);
    return IRQ_HANDLED;
}
```

7.3.2.3 模块初始化（GPIO 读写 + 中断注册）

insmod 加载模块时执行，分两个阶段：先做基础 GPIO 输出/输入读写测试，再配置为输入模式并注册双边沿中断。

```
static int __init gpio_drv_init(void)
{
    int ret, irq_num, val;

    /*
     * 阶段一：GPIO 读写测试
     */

    /* 申请 GPIO 资源 */
    ret = gpio_request(GPIO_TEST_NUM, "gpio_test");
    if (ret < 0) {
        printk(KERN_ERR "gpio_request(%d) failed: %d\n",
               GPIO_TEST_NUM, ret);
        return ret;
    }

    /* 配置为输出模式，初始低电平 */
    gpio_direction_output(GPIO_TEST_NUM, 0);
    printk(KERN_INFO "GPIO%d set to output, initial value: 0\n",
           GPIO_TEST_NUM);

    /* 输出高电平并保持 1 秒 */
    gpio_set_value(GPIO_TEST_NUM, 1);
    mdelay(1000);
    printk(KERN_INFO "GPIO%d set to HIGH\n", GPIO_TEST_NUM);

    /* 切换为输入模式并读取当前电平 */
    gpio_direction_input(GPIO_TEST_NUM);
    val = gpio_get_value(GPIO_TEST_NUM);
    printk(KERN_INFO "GPIO%d switched to input, current value: %d\n",
           GPIO_TEST_NUM, val);

    /*
     * 阶段二：中断配置
     */

    /* 将 GPIO 编号转换为 IRQ 中断号 */
```

（下页继续）

(续上页)

```
    irq_num = gpio_to_irq(GPIO_TEST_NUM);
    if (irq_num < 0) {
        printk(KERN_ERR "gpio_to_irq(%d) failed: %d\n",
               GPIO_TEST_NUM, irq_num);
        gpio_free(GPIO_TEST_NUM);
        return irq_num;
    }
    printk(KERN_INFO "GPIO%d mapped to IRQ %d\n",
           GPIO_TEST_NUM, irq_num);

    /* 注册中断：上升沿 + 下降沿均触发 */
    ret = request_irq(irq_num, gpio_irq_handler,
                      IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING,
                      "gpio_test_irq", GPIO_IRQ_DEV_ID);
    if (ret < 0) {
        printk(KERN_ERR "request_irq(%d) failed: %d\n", irq_num, ret);
        gpio_free(GPIO_TEST_NUM);
        return ret;
    }
    printk(KERN_INFO "IRQ %d registered, waiting for trigger...\n",
           irq_num);

    return 0;
}
```

中断类型标志参考：

标志	说明
IRQF_TRIGGER_RISING	上升沿触发
IRQF_TRIGGER_FALLING	下降沿触发
IRQF_TRIGGER_HIGH	高电平触发
IRQF_TRIGGER_LOW	低电平触发
IRQF_SHARED	共享中断

7.3.2.4 模块卸载（资源释放）

rmmod 卸载模块时执行，依次释放中断和 GPIO 资源。

```
static void __exit gpio_drv_exit(void)
{
    int irq_num = gpio_to_irq(GPIO_TEST_NUM);

    /* 释放中断 */
    free_irq(irq_num, GPIO_IRQ_DEV_ID);
    /* 释放 GPIO */
    gpio_free(GPIO_TEST_NUM);

    printk(KERN_INFO "GPIO test driver unloaded\n");
}
```

7.3.2.5 模块声明

```
module_init(gpio_drv_init);
module_exit(gpio_drv_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Kernel GPIO read/write and interrupt example");
MODULE_AUTHOR("Test");
```

注解:

- 使用 insmod gpio_test.ko 加载模块后，通过 dmesg 查看 GPIO 读写及中断注册日志。
- 外部对该 GPIO 引脚施加电平变化即可在 dmesg 中看到中断触发日志。
- 使用 rmmod gpio_test 卸载模块，资源自动释放。

7.3.3 用户态 GPIO 操作示例

以下为完整的用户态 GPIO 操作示例程序，通过 sysfs 接口实现 GPIO 的导出、方向配置、电平读写及释放。示例以 GPIO1_2（编号 448 + 2 = 450）为例，实际使用时请根据 GPIO 组号对应表修改 GPIO_NUM 宏定义。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define GPIO_NUM 450    /* 示例 GPIO：GPIO1_2 = 448 + 2 */

/* 导出 GPIO，在 /sys/class/gpio/ 下创建 gpioN 节点 */
int gpio_export(int gpio)
{
    FILE *fp = fopen("/sys/class/gpio/export", "w");
    if (fp == NULL) {
        perror("open export failed");
        return -1;
    }
    fprintf(fp, "%d", gpio);
    fclose(fp);
    /* 等待 sysfs 节点创建完成，避免后续访问 direction/value 失败 */
    usleep(100000);
    return 0;
}

/* 释放 GPIO，删除 /sys/class/gpio/gpioN 节点 */
int gpio_unexport(int gpio)
{
    FILE *fp = fopen("/sys/class/gpio/unexport", "w");
    if (fp == NULL) {
        perror("open unexport failed");
```

（下页继续）

(续上页)

```
        return -1;
    }
    fprintf(fp, "%d", gpio);
    fclose(fp);
    return 0;
}

/* 设置 GPIO 方向：dir 取值为 "in" 或 "out" */
int gpio_set_direction(int gpio, const char *dir)
{
    char path[64];
    snprintf(path, sizeof(path), "/sys/class/gpio/gpio%d/direction", gpio);

    FILE *fp = fopen(path, "w");
    if (fp == NULL) {
        perror("open direction failed");
        return -1;
    }
    fprintf(fp, "%s", dir);
    fclose(fp);
    return 0;
}

/* 设置 GPIO 输出电平：value 为 0(低) 或 1(高) */
int gpio_set_value(int gpio, int value)
{
    char path[64];
    snprintf(path, sizeof(path), "/sys/class/gpio/gpio%d/value", gpio);

    FILE *fp = fopen(path, "w");
    if (fp == NULL) {
        perror("open value for write failed");
        return -1;
    }
    fprintf(fp, "%d", value);
    fclose(fp);
    return 0;
}

/* 读取 GPIO 输入电平，读取结果通过 value 返回 */
int gpio_get_value(int gpio, int *value)
{
    char path[64];
    snprintf(path, sizeof(path), "/sys/class/gpio/gpio%d/value", gpio);

    FILE *fp = fopen(path, "r");
    if (fp == NULL) {
        perror("open value for read failed");
        return -1;
    }
    /* sysfs value 文件通常返回字符 '0' 或 '1' */
    char buf[2];
    fread(buf, 1, 1, fp);
    *value = atoi(buf);
```

(下页继续)

(续上页)

```
    fclose(fp);
    return 0;
}

int main(void)
{
    int ret, value;

    /* 步骤1: 导出 GPIO */
    ret = gpio_export(GPIO_NUM);
    if (ret < 0)
        exit(1);
    printf("GPIO%d exported\n", GPIO_NUM);

    /* 步骤2: 设置为输出模式 */
    ret = gpio_set_direction(GPIO_NUM, "out");
    if (ret < 0) {
        gpio_unexport(GPIO_NUM);
        exit(1);
    }
    printf("GPIO%d set to output\n", GPIO_NUM);

    /* 步骤3: 输出高电平并保持 1 秒 */
    ret = gpio_set_value(GPIO_NUM, 1);
    if (ret < 0) {
        gpio_unexport(GPIO_NUM);
        exit(1);
    }
    printf("GPIO%d output HIGH\n", GPIO_NUM);
    usleep(1000000);

    /* 步骤4: 切换为输入模式 */
    ret = gpio_set_direction(GPIO_NUM, "in");
    if (ret < 0) {
        gpio_unexport(GPIO_NUM);
        exit(1);
    }
    printf("GPIO%d set to input\n", GPIO_NUM);

    /* 步骤5: 读取输入电平 */
    ret = gpio_get_value(GPIO_NUM, &value);
    if (ret < 0) {
        gpio_unexport(GPIO_NUM);
        exit(1);
    }
    printf("GPIO%d input value: %d\n", GPIO_NUM, value);

    /* 步骤6: 释放 GPIO 资源 */
    ret = gpio_unexport(GPIO_NUM);
    if (ret < 0)
        exit(1);
    printf("GPIO%d unexported\n", GPIO_NUM);

    return 0;
}
```

8 UART 操作指南

本文档说明在 CV184x 平台上配置与使用 UART（串口）、环回测试、常用波特率设置以及用户态程序开发的基本流程。

8.1 操作准备

8.1.1 内核要求

- 使用官方发布的 SDK 编译后的内核镜像。

8.1.2 设备树中使能 UART

在设备树中将目标 UART 节点的 status 设为 "okay" 。UART 节点定义位于 SDK 的 build/boards/default/dts/cv184x/cv184x_base.dtsi 中。以 UART1 为例：

```
uart1: serial@04150000 {
    compatible = "snps,dw-apb-uart";
    reg = <0x0 0x04150000 0x0 0x1000>;
    clock-frequency = <25000000>;
    reg-shift = <2>;
    reg-io-width = <4>;
    status = "okay";  /* 关闭则设为 "disabled" */
};
```

修改后需重新编译设备树并烧录到设备。

8.1.3 UART 使用 DMA 时的设备树配置

若需使能 UART DMA，需同时修改板级 DTS 与 cv184x_base.dtsi：

- 板级 DTS 路径：build/boards/cv184x/板名/dts_arm/板名.dts（板名替换为实际板卡名）。
- 在板级 DTS 的 sysdma_remap 中为 UART 分配 DMA 通道，并在对应 UART 节点中增加 dmas、dma-names 等属性。

cv184x_base.dtsi 中 UART1 的 DMA 示例（与 sysdma_remap 配合）：

```
sysdma_remap {
    ch-remap = <CVI_I2S0_RX CVI_I2S2_TX CVI_UART1_RX CVI_UART1_TX
        CVI_SPI_NOR_RX CVI_SPI_NOR_TX CVI_I2S2_RX CVI_I2S3_TX>;
};

uart1: serial@04150000 {
    compatible = "snps,dw-apb-uart";
    reg = <0x0 0x04150000 0x0 0x1000>;
    clock-frequency = <25000000>;
    reg-shift = <2>;
    reg-io-width = <4>;
    status = "okay";
    dmas = <&dmac 2 1 1>, <&dmac 3 1 1>;
    dma-names = "rx", "tx";
    capability = "txrx";
};
```

dmas 中的通道号需与 sysdma_remap 中的分配一致，例如 CVI_UART1_RX 对应 dmac 通道 2，CVI_UART1_TX 对应 dmac 通道 3。修改后重新编译设备树并烧录。

8.1.4 引脚复用

通过 cvi_pimux 工具或设备树将所用 GPIO 引脚配置为 UART 功能（TX、RX、CTS、RTS 等），具体引脚以板级原理图为准。

8.2 操作流程

8.2.1 UART 环回测试

用于验证串口收发与引脚是否正常：将 UART 的 TX 与 RX 短接，发送数据后应能从同一串口读回相同内容。

1. **短接 UART1 的 TX 与 RX 引脚。**
2. **在终端执行环回测试命令（以 ttyS1 为例）：**

```
stty -F /dev/ttyS1 115200 cs8 -cstopb -parenb
hexdump -C < /dev/ttyS1 &
echo "LOOPBACK_TEST" > /dev/ttyS1
```

```
[root@cvitek]/mnt/sd# stty -F /dev/ttyS1 115200 cs8 -cstopb -parenb
[root@cvitek]/mnt/sd# hexdump -C < /dev/ttyS1 &
[root@cvitek]/mnt/sd# echo "LOOPBACK_TEST" > /dev/ttyS1
[root@cvitek]/mnt/sd# 00000000  00 4f 4f 50 42 41 43 4b  5f 54 45 53 54 0a 0a 5e  |.OOPBACK_TEST..^|
00000010  40 4f 4f 50 42 41 43 4b  5f 54 45 53 54 0a 0a 0a  |@OOPBACK_TEST...|
00000020  0a 5e 40 4f 4f 50 42 41  43 4b 5f 54 45 53 54 0a  |.^@OOPBACK_TEST.|
00000030  0a 0a 0a 0a 0a 0a 0a 5e  40 4f 4f 50 42 41 43 4b  |.......^@OOPBACK|
00000040  5f 54 45 53 54 0a 0a 0a  0a 0a 0a 0a 0a 0a 0a 0a  |_TEST...........|
00000050  0a 0a 0a 0a 0a 5e 40 4f  4f 50 42 41 43 4b 5f 54  |.....^@OOPBACK_T|
00000060  45 53 54 0a 0a 0a 0a 0a  0a 0a 5e 4f 43 45 0a 0a  |EST.......^OCE..|
00000070  0a 0a 0a 0a 0a 0a 0a 0a  0a 0a 40 50 43 54 54 0a  |..........@PCTT.|
00000080  0a 0a 0a 5e 45 0a 0a 0a  0a 0a 0a 0a 0a 0a 0a 54  |...^E..........T|
00000090  6e 0a 09 00 00 60 fb 74  74 75 0a 54 00 5e fe 9f  |n....`.ttu.T.^..|
000000a0  28 2d 72 23 75 72 50 72  50 0a 0a 50 00 c0 9f 28  |(-r#urPrP..P...(|
000000b0  2d 61 74 20 2c 72 2c 72  0a 00 00 63 61 9f 9f 0a  |-at ,r,r...ca...|
```

若短接正常，hexdump 应能打印出与 echo 一致的内容。

8.2.2 UART 串口通信（终端收发）

设置波特率与帧格式（以 ttyS1、115200 8N1 为例）：

```
stty -F /dev/ttyS1 115200 cs8 -cstopb -parenb
```

发送数据：

```
echo "Hello UART" > /dev/ttyS1
```

接收数据（需在另一终端或另一台机连接该串口）：

```
cat /dev/ttyS1
```

查看当前串口配置：

```
stty -F /dev/ttyS1 -a
```

8.3 UART 扩展操作（常用 / 排障 / 高级）

除上述基础收发、环回测试与查看配置外，以下按 **常用扩展操作** → **排障类操作** → **高级配置操作** 的顺序补充串口调试、参数固化、流控、数据捕获与排障等说明。示例中设备节点以 /dev/ttyS1 为例，请按实际串口替换。

8.3.1 常用扩展操作（基础功能补充）

1. 串口流控配置（缓解大流量丢包）

若串口传输大文件或高频数据时出现丢包，可尝试开启硬件或软件流控。硬件流控需板级支持 RTS/CTS 引脚并正确连接。

```
# 开启硬件流控（RTS/CTS）
stty -F /dev/ttyS1 crtscts 115200 cs8

# 开启软件流控（XON/XOFF）
stty -F /dev/ttyS1 ixon ixoff 115200 cs8

# 关闭所有流控（默认）
stty -F /dev/ttyS1 -crtscts -ixon -ixoff
```

硬件流控可靠性更高，适合工业等对稳定性要求高的场景；软件流控无需额外引脚，适合简单场景。

2. 串口数据捕获与分析（调试用）

需要抓取串口收发的原始数据用于调试协议时，可采用以下方式。

用 cat + tee 保存接收数据

```
cat /dev/ttyS1 | tee /tmp/uart_recv.log  # 实时查看并保存
```

3. 串口发送 / 接收二进制数据（非文本）

前文 echo 仅适合发送文本。传输二进制文件（如固件、数据包）时建议使用 cat/dd，避免换行或转义影响数据。

```
# 发送二进制文件到串口
cat /mnt/sd/data.bin > /dev/ttyS1
# 文件根据实际修改

# 从串口接收二进制数据并保存（按块读取）
dd if=/dev/ttyS1 of=/tmp/recv_bin.bin bs=1024 count=10  # 读取 10KB
```

8.3.2 排障类操作（解决串口异常）

1. 检查串口硬件与占用状态

```
# 查看串口设备是否存在
ls -l /dev/ttyS*

# 查看串口驱动状态（占用、中断、错误计数等，嵌入式常用）
cat /proc/tty/driver/serial

# 检查串口是否被其他进程占用
lsof /dev/ttyS1  # 有输出则表示被占用，可结束对应进程
```

2. 清除串口缓存 / 重置串口

串口出现乱码、卡死时，可尝试清空缓冲或恢复默认配置。

```
# 清空串口输入/输出缓冲
stty -F /dev/ttyS1 flush 0   # 清空输入缓存
stty -F /dev/ttyS1 flush 1   # 清空输出缓存
```

（下页继续）

(续上页)

```
# 将串口恢复为合理默认配置
stty -F /dev/ttyS1 sane
```

8.3.3 高级配置操作（适配特殊场景）

1. 配置串口奇偶校验

前文示例为无校验（-parenb）。若外设要求奇偶校验，可按需设置。

```
# 偶校验（even parity）
stty -F /dev/ttyS1 115200 cs8 -cstopb parenb -parodd

# 奇校验（odd parity）
stty -F /dev/ttyS1 115200 cs8 -cstopb parenb parodd

# 无校验（默认，与前文一致）
stty -F /dev/ttyS1 115200 cs8 -cstopb -parenb
```

2. 调整串口输入 / 输出与回显（缓解粘包等）

```
# 设置输出延迟等（单位 0.01 秒量级），可减少小包发送时的粘包
stty -F /dev/ttyS1 ospeed 115200 ocrnl onlcr -ocrnl -onlcr delay 1

# 关闭输入回显（交互时避免本地回显重复）
stty -F /dev/ttyS1 -echo
```

8.3.4 UART 扩展操作小结

- **常用配置**：永久保存串口参数（rc.local 或 systemd）、配置流控（减少丢包）、使用 cat/dd 传输二进制数据。
- **排障操作**：检查串口设备与占用（ls、/proc/tty/driver/serial、lsof/fuser）、清空缓冲与重置（stty flush/sane）、循环收发验证稳定性。
- **高级调试**：数据捕获（tee、screen、minicom）、奇偶校验、输出/回显与延迟、串口控制台绑定。

注意：串口参数（波特率、流控、校验、数据位/停止位）需与对端外设完全一致，否则易出现乱码或丢包；调试时优先使用 screen 或 minicom 等工具；二进制数据传输避免使用 echo，使用 cat/dd 更可靠。

8.4 UART 特定波特率输出（以 3M 为例）

当需要 3M 等非标准波特率时，需修改内核中波特率表与 termbits，并保证设备树中 UART 时钟足够大，否则会出现乱码或无法工作。

步骤 1：修改内核波特率表与 termbits

1. **修改 linux_5.10/drivers/tty/tty_baudrate.c**

   在 baud_table 和 baud_bits 中增加 3000000（若需其他波特率，将 3000000 替换为目标值）。示例片段：

```
static const speed_t baud_table[] = {
    0, 50, 75, 110, 134, 150, 200, 300, 600, 1200, 1800, 2400,
    4800, 9600, 19200, 38400, 57600, 115200, 230400, 460800,
#ifdef __sparc__
    76800, 153600, 307200, 614400, 921600, 500000, 576000,
    1000000, 1152000, 1500000, 2000000
#else
    500000, 576000, 921600, 1000000, 1152000, 1500000, 2000000,
    3125000, 3000000, 6125000, 12500000
#endif
};

static const tcflag_t baud_bits[] = {
    B0, B50, B75, B110, B134, B150, B200, B300, B600, B1200, B1800, B2400,
    B4800, B9600, B19200, B38400, B57600, B115200, B230400, B460800,
#ifdef __sparc__
    B76800, B153600, B307200, B614400, B921600, B500000, B576000,
    B1000000, B1152000, B1500000, B2000000
#else
    B500000, B576000, B921600, B1000000, B1152000, B1500000, B2000000,
    B3125000, B3000000, B6125000, B12500000
#endif
};
```

```
    500000, 576000, 921600, 1000000, 1152000, 1500000, 2000000,
    3125000, 4000000, 6125000, 12500000
    500000, 576000, 921600, 1000000, 1152000, 1500000, 2000000,
    2500000, 3000000, 6125000, 12500000
```

```
    B500000, B576000, B921600, B1000000, B1152000, B1500000, B2000000,
    B3125000, B4000000, B6125000, B12500000
    B500000, B576000, B921600, B1000000, B1152000, B1500000, B2000000,
    B2500000, B3000000, B6125000, B12500000
```

2. **修改 linux_5.10/include/uapi/asm-generic/termbits.h**

   - 将 3M 波特率注释打开，并注释掉 4000000 波特率（如需修改其他波特率则将 3000000 替换为其他数值）。

```
#define  B3000000 0010015
// #define  B4000000 0010015
```

   - 注释掉重复的 B3000000 定义（即数值为 0020001 的那一行），保留与 baud_table 一致的定义。

```
// #define B3000000 0020001
```

修改后重新编译内核。

步骤 2：设备树中 UART 时钟

若环回测试通过但实际通信无输出或乱码，多为 UART 输入时钟（CLK）偏小。UART 时钟建议不小于 波特率 × 16 。在设备树中将该 UART 的 clock-frequency 调大（例如改为 187500000 即 187.5MHz），重新编译设备树并烧录。

```
    uart1: serial@04150000 {
        compatible = "snps,dw-apb-uart";
        reg = <0x0 0x04150000 0x0 0x1000>;
        clock-frequency = <187500000>;
        reg-shift = <2>;
        reg-io-width = <4>;
        status = "okay";// 关闭设为"disabled"
#if 0
        dmas = <&dmac 2 1 1
        &dmac 3 1 1>;
        dma-names = "rx", "tx";
        capability = "txrx";
#endif
    };
```

步骤 3：验证 3M 波特率是否正常

按本文「操作流程」中的 **UART 环回测试**与 **UART 串口通信（终端收发）**进行 UART 收发测试：先将目标 UART 的 TX 与 RX 短接做环回测试，或将两台设备通过串口相连做收发测试；在 stty 中设置波特率为 3000000（例如 stty -F /dev/ttyS1 3000000 cs8 -cstopb -parenb），发送并读回数据。若环回/对端接收内容与发送一致，则 3M 波特率工作正常。

8.5 用户态程序开发示例

以下示例演示用户空间串口收发流程：打开串口、配置波特率与 8N1、执行读写测试。

示例内容：

- uart_config：8N1、可配置波特率、原始模式、读超时 10 秒
- uart_write / uart_read：发送与接收
- main：循环收发

请按实际设备节点替换 /dev/ttyS1。若内核未支持 3M 波特率，请将 B3000000 改为 B115200 等（参见本文「UART 特定波特率输出（以 3M 为例）」小节）。

基本流程：打开设备、配置 termios、收发、关闭

```c
#include <fcntl.h>
#include <termios.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>

/* 配置串口参数（8N1，波特率由参数指定） */
int uart_config(int fd, speed_t baudrate) {
    struct termios tty;
    memset(&tty, 0, sizeof(tty));

    if (tcgetattr(fd, &tty) != 0) {
        printf("获取串口配置失败: %s\n", strerror(errno));
        return -1;
    }

    cfsetispeed(&tty, baudrate);
    cfsetospeed(&tty, baudrate);

    tty.c_cflag &= ~PARENB;
    tty.c_cflag &= ~PARODD;
    tty.c_cflag &= ~CSTOPB;
    tty.c_cflag &= ~CSIZE;
    tty.c_cflag |= CS8;
    tty.c_cflag &= ~CRTSCTS;
    tty.c_cflag |= CREAD | CLOCAL;

    tty.c_lflag &= ~ICANON;
    tty.c_lflag &= ~ECHO;
    tty.c_lflag &= ~ECHOE;
    tty.c_lflag &= ~ISIG;

    tty.c_iflag &= ~(IXON | IXOFF | IXANY);
    tty.c_iflag &= ~(IGNBRK|BRKINT|PARMRK|ISTRIP|INLCR|IGNCR|ICRNL);

    tty.c_oflag &= ~OPOST;
    tty.c_oflag &= ~ONLCR;

    tty.c_cc[VTIME] = 100;  /* 10 秒超时 */
    tty.c_cc[VMIN] = 1;

    if (tcsetattr(fd, TCSANOW, &tty) != 0) {
        printf("设置串口配置失败: %s\n", strerror(errno));
        return -1;
    }
    tcflush(fd, TCIOFLUSH);
    return 0;
}

ssize_t uart_write(int fd, const char *data, size_t len) {
    if (fd < 0 || !data || len == 0) return -1;
    ssize_t ret = write(fd, data, len);
    if (ret < 0) {
        printf("写数据失败: %s\n", strerror(errno));
```

（下页继续）

(续上页)

```
        return -1;
    }
    printf("发送成功: %.*s (长度: %zd字节)\n", (int)ret, data, ret);
    return ret;
}

ssize_t uart_read(int fd, char *buf, size_t buf_len) {
    if (fd < 0 || !buf || buf_len == 0) return -1;
    memset(buf, 0, buf_len);
    ssize_t ret = read(fd, buf, buf_len - 1);
    if (ret < 0) {
        printf("读数据失败: %s\n", strerror(errno));
        return -1;
    } else if (ret == 0) {
        printf("读数据超时 (10秒未收到数据) \n");
        return 0;
    }
    buf[ret] = '\0';
    printf("接收成功: %s (长度: %zd字节)\n", buf, ret);
    return ret;
}

int main(void) {
    int fd = open("/dev/ttyS1", O_RDWR | O_NOCTTY);
    if (fd < 0) {
        printf("打开串口ttyS1失败: %s\n", strerror(errno));
        return -1;
    }
    printf("成功打开ttyS1, 文件描述符: %d\n", fd);

    if (uart_config(fd, B3000000) != 0) {
        close(fd);
        return -1;
    }
    printf("串口配置完成 (3000000 8N1) \n");

    char send_buf[64] = "Hello ttyS1! This is UART test.\n";
    char recv_buf[128];
    while (1) {
        uart_write(fd, send_buf, strlen(send_buf));
        uart_read(fd, recv_buf, sizeof(recv_buf));
        sleep(1);
    }

    close(fd);
    return 0;
}
```

运行方式说明

1. **保存源码**: 将上述代码保存为文件, 例如 uart_demo.c。
2. **编译**: 在宿主机或目标板上使用对应工具链编译。

```
# 交叉编译示例 (以实际工具链为准)
```

(下页继续)

(续上页)

```
arm-none-linux-uclibcgnueabihf-gcc -o uart_demo uart_demo.c -static
```

3. **运行**：将可执行文件拷贝到开发板后，在目标板上执行。需对串口设备节点（如 /dev/ttyS1）有读写权限。

```
./uart_demo
```

   程序会循环发送 Hello ttyS1! This is UART test. 并阻塞等待接收（最多 10 秒），可使用串口工具打开 ttyS1 对应的串口并发送数据。

4. **修改设备与波特率**：源码中 open("/dev/ttyS1", ...) 可改为实际串口节点；uart_config(fd, B3000000) 可改为 B115200、B921600 等（需内核与设备树支持该波特率，参见本文「高波特率配置」）。

9 Watchdog 操作指南

本文档说明在 CV184x 平台上通过 Linux 标准 Watchdog 框架操作看门狗，包括驱动配置、常用 ioctl 及示例代码。

9.1 操作准备

1. 内核要求

使用 SDK 发布的内核，并确保以下配置已开启（驱动随内核编译，无需 insmod）：

```
CONFIG_WATCHDOG=y
CONFIG_WATCHDOG_CORE=y
CONFIG_DW_WATCHDOG=y
```

2. 使用方式

在控制台通过命令行操作 /dev/watchdog 设备，或在内核态/用户态程序中通过 open、ioctl、write、close 等接口操作看门狗即可。

9.2 操作流程

典型使用流程如下：

1. **打开设备**：打开 /dev/watchdog，打开即启动看门狗，硬件开始计时。
2. **（可选）设置超时**：通过 WDIOC_SETTIMEOUT 设置超时时间（秒）；不设置则使用驱动默认值（42 秒）。
3. **立即喂狗**：打开后应尽快调用 WDIOC_KEEPALIVE 进行一次喂狗，避免在默认/设定超时内未喂狗导致重启。
4. **周期喂狗**：在业务运行期间，在超时时间到达前定期调用 WDIOC_KEEPALIVE 喂狗；喂狗间隔须小于当前超时时间。
5. **关闭设备**：不再需要看门狗时，按顺序执行：先 WDIOC_SETOPTIONS 传入 WDIOS_DISABLECARD 禁用看门狗，再 write(fd, "V", 1) 写入魔术字符，最后 close(fd)；否则关闭文件描述符后硬件仍会继续计时，超时将触发重启。

需要查询当前超时或剩余时间时，可在上述流程中穿插使用 WDIOC_GETTIMEOUT、WDIOC_GETTIMELEFT。

9.3 操作示例

概述

- Watchdog 采用标准 Linux 看门狗框架，提供硬件看门狗。用户通过打开设备、设置超时时间、定期喂狗（keepalive）或关闭设备即可使用。
- 当看门狗超时未被喂狗时，系统将重启。
- **默认状态**：看门狗默认关闭，由用户决定是否开启。
- **可设定的超时时间**（单位：秒）：1、2、5、10、21、42、85。若设定的值不在上述列表，驱动会选用 **大于等于**该值的最接近项；例如设定 8 秒时，实际采用 10 秒。若未设定超时，驱动默认使用 42 秒。

以下示例需包含头文件 <linux/watchdog.h>，ioctl 宏（如 WDIOC_SETTIMEOUT、WDIOC_KEEPALIVE）由该头文件提供，无需自行定义。

9.3.1 打开 Watchdog 并立即喂狗

打开 /dev/watchdog 即启动看门狗，硬件开始计时。**打开后应尽快进行一次喂狗**，否则在超时时间内未喂狗会导致系统重启。

```
/* 需包含头文件：stdio.h, stdlib.h, unistd.h, fcntl.h, string.h, errno.h, sys/ioctl.h, linux/watchdog.h */
int wdt_fd;
int ret;

wdt_fd = open("/dev/watchdog", O_WRONLY);
if (wdt_fd < 0) {
    printf("打开 /dev/watchdog 失败: %s\n", strerror(errno));
    return -1;
}
printf("[步骤 1] 打开 /dev/watchdog 成功, fd=%d\n", wdt_fd);

ret = ioctl(wdt_fd, WDIOC_KEEPALIVE, 0);
if (ret != 0)
    printf("警告: 打开后立即喂狗 ioctl 返回 %d: %s\n", ret, strerror(errno));
else
    printf("      打开后已执行一次喂狗 (WDIOC_KEEPALIVE)\n\n");
```

9.3.2 设置超时时间

通过 ioctl WDIOC_SETTIMEOUT 设置超时时间，单位为秒。可设置值为：1、2、5、10、21、42、85；传入其他值时，驱动会取大于等于该值的最接近支持值。

```
int timeout = 10;  /* 单位：秒，支持 1/2/5/10/21/42/85 */
int ret = ioctl(wdt_fd, WDIOC_SETTIMEOUT, &timeout);
if (ret != 0) {
    printf("[步骤 2] 设置超时失败: %s\n", strerror(errno));
} else {
```

（下页继续）

(续上页)

```
    printf("[步骤 2] 设置超时 %d 秒成功，驱动实际采用: %d 秒\n\n", 10, timeout);
}
```

9.3.3 喂狗（Keepalive）

在超时时间到达前，需定期调用 WDIOC_KEEPALIVE 喂狗，否则看门狗超时将触发系统重启。喂狗间隔应小于当前设定的超时时间。

```
const int keepalive_loops = 5;   /* 演示喂狗次数，可改为 while(running) 等 */
int i;

printf("[步骤 3] 开始周期性喂狗（共 %d 次，每次间隔 1 秒）\n", keepalive_loops);
for (i = 0; i < keepalive_loops; i++) {
    ioctl(wdt_fd, WDIOC_KEEPALIVE, 0);
    printf("      第 %d 次喂狗\n", i + 1);
    sleep(1);
}
printf("\n");
```

9.3.4 获取当前超时时间

通过 WDIOC_GETTIMEOUT 可查询驱动当前采用的超时时间（秒）。

```
int timeout_sec = 0;
if (ioctl(wdt_fd, WDIOC_GETTIMEOUT, &timeout_sec) == 0) {
    printf("[步骤 4] 当前超时时间: %d 秒\n", timeout_sec);
} else {
    printf("[步骤 4] 获取超时时间失败（可选）\n");
}
```

9.3.5 获取剩余时间

若驱动支持，可通过 WDIOC_GETTIMELEFT 查询距离下次超时还剩多少秒，便于调试或监控。

```
int timeleft_sec = 0;
if (ioctl(wdt_fd, WDIOC_GETTIMELEFT, &timeleft_sec) == 0) {
    printf("[步骤 5] 距离下次超时剩余: %d 秒\n\n", timeleft_sec);
} else {
    printf("[步骤 5] 获取剩余时间不支持或失败（可选）\n\n");
}
```

9.3.6 关闭 Watchdog（Magic Close）

本驱动支持 **Magic Close**：在关闭设备前，必须先 **禁用看门狗**，再向设备写入魔术字符 'V'，最后才能 close。否则即使关闭了文件描述符，硬件看门狗仍会继续计时，超时未喂狗将导致系统重启。

正确顺序：WDIOS_DISABLECARD → write(..., "V", 1) → close。

```
int option = WDIOS_DISABLECARD;
int ret;

printf("[步骤 6] 关闭 Watchdog (Magic Close)\n");
ret = ioctl(wdt_fd, WDIOC_SETOPTIONS, &option);
if (ret != 0)
    printf("      警告: WDIOC_SETOPTIONS WDIOS_DISABLECARD 返回 %d: %s\n", ret,␣
↪strerror(errno));

if (wdt_fd >= 0) {
    if (write(wdt_fd, "V", 1) != 1)
        printf("      警告: 写入魔术字符 'V' 失败: %s\n", strerror(errno));
    else
        printf("      已写入魔术字符 'V'\n");
    close(wdt_fd);
    wdt_fd = -1;
    printf("      已 close 设备\n");
}
printf("\n程序正常结束\n");
```

9.3.7 完整用户态程序示例

以下为将上述各步骤串联而成的完整 C 程序，可直接编译运行。需包含头文件 <linux/watchdog.h>，ioctl 宏由该头文件提供。

```
1  /*
2   * Watchdog 操作示例（遵循标准 Linux 看门狗框架）
3   * 需包含头文件 <linux/watchdog.h>，ioctl 宏由该头文件提供。
4   */
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <unistd.h>
8  #include <fcntl.h>
9  #include <string.h>
10 #include <errno.h>
11 #include <sys/ioctl.h>
12 #include <linux/watchdog.h>
13
14 int main(void)
15 {
16     int wdt_fd;
17     int timeout;
18     int timeout_sec;
19     int timeleft_sec;
```

（下页继续）

(续上页)

```
20  int option;
21  int ret;
22  int i;
23  const int keepalive_loops = 5;  /* 演示喂狗次数，可改为 while(running) 等 */
24
25  printf("=== Watchdog 操作示例 ===\n\n");
26
27  /* 步骤 1: 打开 Watchdog 并立即喂狗 */
28  wdt_fd = open("/dev/watchdog", O_WRONLY);
29  if (wdt_fd < 0) {
30      printf("打开 /dev/watchdog 失败: %s\n", strerror(errno));
31      return -1;
32  }
33  printf("[步骤 1] 打开 /dev/watchdog 成功, fd=%d\n", wdt_fd);
34
35  ret = ioctl(wdt_fd, WDIOC_KEEPALIVE, 0);
36  if (ret != 0)
37      printf("警告: 打开后立即喂狗 ioctl 返回 %d: %s\n", ret, strerror(errno));
38  else
39      printf("      打开后已执行一次喂狗 (WDIOC_KEEPALIVE)\n\n");
40
41  /* 步骤 2: 设置超时时间（单位: 秒，支持 1/2/5/10/21/42/85） */
42  timeout = 10;
43  ret = ioctl(wdt_fd, WDIOC_SETTIMEOUT, &timeout);
44  if (ret != 0) {
45      printf("[步骤 2] 设置超时失败: %s\n", strerror(errno));
46  } else {
47      printf("[步骤 2] 设置超时 %d 秒成功，驱动实际采用: %d 秒\n\n", 10, timeout);
48  }
49
50 /* 步骤 3: 设置超时后再次喂狗，并开始周期性喂狗 */
51 ret = ioctl(wdt_fd, WDIOC_KEEPALIVE, 0);
52 if (ret != 0) {
53     printf("[步骤 3] 设置超时后立即喂狗失败: %s\n", strerror(errno));
54 } else {
55     printf("[步骤 3] 设置超时后已立即喂狗一次\n");
56 }
57
58 printf("[步骤 3] 开始周期性喂狗（共 %d 次，每次间隔 1 秒）\n", keepalive_loops);
59 for (i = 0; i < keepalive_loops; i++) {
60     ret = ioctl(wdt_fd, WDIOC_KEEPALIVE, 0);
61     if (ret != 0) {
62         printf("      第 %d 次喂狗失败: %s\n", i + 1, strerror(errno));
63         break;
64     }
65     printf("      第 %d 次喂狗成功\n", i + 1);
66     sleep(1);
67 }
68 printf("\n");
69
70 /* 步骤 4: 获取当前超时时间 */
71 timeout_sec = 0;
72 if (ioctl(wdt_fd, WDIOC_GETTIMEOUT, &timeout_sec) == 0) {
73     printf("[步骤 4] 当前超时时间: %d 秒\n", timeout_sec);
```

(下页继续)

(续上页)

```
    } else {
        printf("[步骤 4] 获取超时时间失败（可选）\n");
    }

    /* 步骤 5: 获取剩余时间（可选，部分驱动支持） */
    timeleft_sec = 0;
    if (ioctl(wdt_fd, WDIOC_GETTIMELEFT, &timeleft_sec) == 0) {
        printf("[步骤 5] 距离下次超时剩余: %d 秒\n\n", timeleft_sec);
    } else {
        printf("[步骤 5] 获取剩余时间不支持或失败（可选）\n\n");
    }

    /* 步骤 6: 关闭 Watchdog（Magic Close）*/
    /* 正确顺序: WDIOS_DISABLECARD -> write("V", 1) -> close */
    printf("[步骤 6] 关闭 Watchdog (Magic Close)\n");
    option = WDIOS_DISABLECARD;
    ret = ioctl(wdt_fd, WDIOC_SETOPTIONS, &option);
    if (ret != 0)
        printf("      警告: WDIOC_SETOPTIONS WDIOS_DISABLECARD 返回 %d: %s\n", ret,␣
↪strerror(errno));

    if (wdt_fd >= 0) {
        if (write(wdt_fd, "V", 1) != 1)
            printf("      警告: 写入魔术字符 'V' 失败: %s\n", strerror(errno));
        else
            printf("      已写入魔术字符 'V'\n");
        close(wdt_fd);
        wdt_fd = -1;
        printf("      已 close 设备\n");
    }
    printf("\n程序正常结束\n");
    return 0;
}
```

9.3.8 命令行快速示例

以下说明如何在命令行下实现与上文「完整用户态程序示例」相同的功能。**注意**：打开 /dev/watchdog 后若不定期喂狗，系统会在超时后重启，请仅在测试环境使用。设置超时（WDIOC_SETTIMEOUT）、查询超时/剩余时间（WDIOC_GETTIMEOUT / WDIOC_GETTIMELEFT）需通过 C 程序或等价工具完成；推荐直接编译并运行上文的完整用户态程序以一次性完成所有步骤。

推荐：一次性完成所有功能

编译并运行上文「完整用户态程序示例」中的完整 C 程序，可依次完成：打开并立即喂狗、设置超时、获取当前超时与剩余时间、周期性喂狗、Magic Close 关闭。

```
# 编译（示例：假设源文件为 watchdog_demo.c，编译工具为 arm-none-linux-uclibcgnueabihf-gcc）
arm-none-linux-uclibcgnueabihf-gcc -o watchdog_demo watchdog_demo.c -static
# 将运行文件拷贝到开发板，并运行
./watchdog_demo
```

步骤 1：手动激活看门狗（无其他进程干扰）

在无其他进程占用或喂狗的前提下，手动启动喂狗进程，使看门狗开始计时并在超时前被持续喂狗。

```
# 方式 1：用 watchdog 工具（10 秒超时，后台运行）
watchdog -t 10 /dev/watchdog &

# 方式 2：无工具则用 Shell 循环
( while true; do echo -n '.' > /dev/watchdog; sleep 1; done ) &
```

步骤 2：关闭 Watchdog（Magic Close）

```
killall watchdog   # 自带 magic close 功能，会自动执行关闭看门狗操作
```

10 PWM 操作指南

本文档说明在 CV184x 平台上通过 sysfs 配置与使用 PWM、引脚复用方法以及命令行与用户态程序示例。

10.1 操作准备

1. 内核要求

- 使用 SDK 提供的内核，并确保内核配置中已开启 CONFIG_PWM （驱动随内核自动加载，无需 insmod）。

2. 引脚复用

PWM 信号需通过 pinmux 将对应 GPIO 引脚配置为 PWM 功能。在目标板上执行 cvi_pinmux 命令时，请以实际使用的 PWM 通道及原理图为准。

示例：将 PWM6 所用引脚从 JTAG 切到 PWM

```
cvi_pinmux -r JTAG_CPU_TCK
cvi_pinmux -w JTAG_CPU_TCK/PWM_6
```

示例：将 PWM14 所用引脚从 I2C 切到 PWM

```
cvi_pinmux -r IIC2_SCL
cvi_pinmux -w IIC2_SCL/PWM_14
```

10.2 操作流程

使用方式

- 在控制台通过 shell 命令操作 sysfs 接口；
- 或在用户态/内核态程序中通过读写 sysfs 节点操作 PWM。

硬件与 sysfs 对应关系

- 时钟频率：250 MHz。
- 通道数量：16 路，每路可独立配置周期与占空比。

- 控制器与 sysfs 路径：
  - 芯片内共有 3 个 PWM 控制器，对应 sysfs 中的 pwmchip0、pwmchip6、pwmchip12。
  - 原理图上的 pwm0 ~ pwm15 与控制器、sysfs 的对应关系如下：
    - pwm0 ~ pwm5 → /sys/class/pwm/pwmchip0/pwm0 ~ pwm5
    - pwm6 ~ pwm11 → /sys/class/pwm/pwmchip6/pwm0 ~ pwm5
    - pwm12 ~ pwm15 → /sys/class/pwm/pwmchip12/pwm0 ~ pwm3

  即 pwmchip6 的 pwm0 ~ pwm5 对应原理图 pwm6 ~ pwm11，pwmchip12 的 pwm0 ~ pwm3 对应原理图 pwm12 ~ pwm15。

10.3 操作示例

10.3.1 命令行操作示例

以下以 pwmchip0 的 pwm1、pwmchip6 的 pwm0、pwmchip12 的 pwm2 为例（对应原理图上的 pwm1、pwm6、pwm14）。实际使用时请按需替换控制器编号与通道号。

步骤 1：导出 PWM 通道

```
echo 1 > /sys/class/pwm/pwmchip0/export
echo 0 > /sys/class/pwm/pwmchip6/export
echo 2 > /sys/class/pwm/pwmchip12/export
```

步骤 2：设置周期（单位：纳秒）

```
echo 1000000 > /sys/class/pwm/pwmchip0/pwm1/period
echo 1000000 > /sys/class/pwm/pwmchip6/pwm0/period
echo 1000000 > /sys/class/pwm/pwmchip12/pwm2/period
```

步骤 3：设置占空比（单位：纳秒）

```
echo 500000 > /sys/class/pwm/pwmchip0/pwm1/duty_cycle   # 500000/1000000 = 50%
echo 300000 > /sys/class/pwm/pwmchip6/pwm0/duty_cycle
echo 500000 > /sys/class/pwm/pwmchip12/pwm2/duty_cycle
```

步骤 4：使能 PWM 输出

使能后可在对应引脚用示波器或外设观察波形。

```
echo 1 > /sys/class/pwm/pwmchip0/pwm1/enable
echo 1 > /sys/class/pwm/pwmchip6/pwm0/enable
echo 1 > /sys/class/pwm/pwmchip12/pwm2/enable
```

步骤 5：禁用 PWM 输出

```
echo 0 > /sys/class/pwm/pwmchip0/pwm1/enable
echo 0 > /sys/class/pwm/pwmchip6/pwm0/enable
echo 0 > /sys/class/pwm/pwmchip12/pwm2/enable
```

步骤 6：取消导出 PWM 通道

```
echo 1 > /sys/class/pwm/pwmchip0/unexport
echo 0 > /sys/class/pwm/pwmchip6/unexport
echo 2 > /sys/class/pwm/pwmchip12/unexport
```

步骤 7：查看 PWM 通道状态

以下以 pwmchip0/pwm1 为例，可查看周期、占空比、使能状态及控制器信息：

```
# 查看 pwmchip0/pwm1 的周期
cat /sys/class/pwm/pwmchip0/pwm1/period
# 查看 pwmchip0/pwm1 的占空比
cat /sys/class/pwm/pwmchip0/pwm1/duty_cycle
# 查看 pwmchip0/pwm1 是否使能（1=使能，0=禁用）
cat /sys/class/pwm/pwmchip0/pwm1/enable
# 查看 PWM 控制器的基本信息
cat /sys/class/pwm/pwmchip0/device/uevent
```

10.3.2 最高与最低频率操作示例

以下说明如何配置 PWM 的最高频率与最低频率，并给出操作步骤与典型参数。

采用示波器观测波形时，需保证示波器采样率至少大于被测信号频率的两倍，否则无法正确还原波形。

最高频率

- 本设备最高支持 50 MHz，对应周期为 20 ns（1/50 MHz = 20 ns）。
- 为得到有效波形，占空比时间（duty_ns）必须小于周期时间，通常采用 50% 占空比测试。

以 pwmchip0/pwm1 为例：

```
echo 1 > /sys/class/pwm/pwmchip0/export
echo 20 > /sys/class/pwm/pwmchip0/pwm1/period      # 周期 20 ns → 频率 50 MHz
echo 10 > /sys/class/pwm/pwmchip0/pwm1/duty_cycle  # 50% 占空比：20 × 0.5 = 10 ns
echo 1 > /sys/class/pwm/pwmchip0/pwm1/enable
```

结果：输出频率约 50 MHz，占空比 50%，可用示波器在对应引脚验证。

最低频率

- 由于内核中周期与占空比使用有符号整数（ns）表示，周期最大可取 2147483646 ns（避免溢出）。
- 为生成有效波形，占空比时间（duty_ns）必须小于周期时间，通常选择 50% 占空比测试：duty_ns = period_ns × 50% = 2147483646 × 0.5 = 1073741823 ns。

以 pwmchip0/pwm1 为例：

```
echo 1 > /sys/class/pwm/pwmchip0/export
echo 2147483646 > /sys/class/pwm/pwmchip0/pwm1/period     # 周期 2147483646 ns
echo 1073741823 > /sys/class/pwm/pwmchip0/pwm1/duty_cycle # 50% 占空比
echo 1 > /sys/class/pwm/pwmchip0/pwm1/enable
```

结果：输出周期约 2.147 s（频率约 0.466 Hz），占空比 50%。可用示波器观察长周期方波。

10.3.3 用户态程序示例

以下以 pwmchip6 的 pwm0 为例（对应原理图 pwm6），演示在用户态通过 open/write 操作 sysfs 完成导出、设置周期与占空比、使能/禁用输出、取消导出。路径与通道号可通过宏 PWM_PATH、PWM_NUM 修改。

辅助函数：向 sysfs 节点写入字符串

```
#define PWM_PATH "/sys/class/pwm/pwmchip6"
#define PWM_NUM  0

int pwm_write(const char *path, const char *value) {
    int fd = open(path, O_WRONLY);
    if (fd < 0) {
        perror("打开文件失败");
        return -1;
    }
    if (write(fd, value, strlen(value)) < 0) {
        perror("写入失败");
        close(fd);
        return -1;
    }
    close(fd);
    return 0;
}
```

步骤 1：导出 PWM 通道

```
char path[256];
snprintf(path, sizeof(path), "%s/export", PWM_PATH);
if (pwm_write(path, "1") < 0)   /* 导出 pwm0 */
    return -1;
usleep(100000);   /* 等待约 100ms 再操作该通道 */
```

步骤 2：配置周期（单位：纳秒，示例 1000000 ns）

```
snprintf(path, sizeof(path), "%s/pwm%d/period", PWM_PATH, PWM_NUM);
if (pwm_write(path, "1000000") < 0)
    return -1;
```

步骤 3：配置占空比（单位：纳秒，示例 300000 ns）

```
snprintf(path, sizeof(path), "%s/pwm%d/duty_cycle", PWM_PATH, PWM_NUM);
if (pwm_write(path, "300000") < 0)
    return -1;
```

步骤 4：使能 PWM 输出

```
snprintf(path, sizeof(path), "%s/pwm%d/enable", PWM_PATH, PWM_NUM);
if (pwm_write(path, "1") < 0)
    return -1;
```

步骤 5：运行一段时间后禁用 PWM 输出

```
sleep(10);  /* 输出 10 秒 */
if (pwm_write(path, "0") < 0)  /* path 仍为 enable 节点 */
    return -1;
```

步骤 6：取消导出 PWM 通道

```
snprintf(path, sizeof(path), "%s/unexport", PWM_PATH);
pwm_write(path, "1");  /* 取消导出 pwm0，返回值可按需检查 */
```

完整用户态程序（全部代码）

将上述各步骤串联，并加上头文件与 main，即可得到可直接编译运行的程序：

```
// pwm_control.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>

#define PWM_PATH "/sys/class/pwm/pwmchip6"
#define PWM_NUM  0

int pwm_write(const char *path, const char *value) {
    int fd, ret;

    fd = open(path, O_WRONLY);
    if (fd < 0) {
        perror("打开文件失败");
        return -1;
    }

    ret = write(fd, value, strlen(value));
    if (ret < 0) {
        perror("写入失败");
        close(fd);
        return -1;
    }

    close(fd);
    return 0;
}

int main(void) {
    char path[256];
    int ret;

    printf("=== PWM控制程序 ===\n");

    /* 1. 导出 PWM 通道 */
    printf("1. 导出PWM通道 pwm%d...\n", PWM_NUM);
    snprintf(path, sizeof(path), "%s/export", PWM_PATH);
    ret = pwm_write(path, "1");
    if (ret < 0) return -1;
```

（下页继续）

(续上页)

```
    printf("导出成功\n");
    usleep(100000);

    /* 2. 设置周期（1 kHz） */
    printf("2. 设置周期为1kHz...\n");
    snprintf(path, sizeof(path), "%s/pwm%d/period", PWM_PATH, PWM_NUM);
    ret = pwm_write(path, "1000000");
    if (ret < 0) return -1;
    printf("周期设置成功\n");

    /* 3. 设置占空比（30%%） */
    printf("3. 设置占空比为30%%...\n");
    snprintf(path, sizeof(path), "%s/pwm%d/duty_cycle", PWM_PATH, PWM_NUM);
    ret = pwm_write(path, "300000");
    if (ret < 0) return -1;
    printf("占空比设置成功\n");

    /* 4. 启用 PWM */
    printf("4. 启用PWM输出...\n");
    snprintf(path, sizeof(path), "%s/pwm%d/enable", PWM_PATH, PWM_NUM);
    ret = pwm_write(path, "1");
    if (ret < 0) return -1;
    printf("PWM已启用\n");

    /* 5. 运行 10 秒 */
    printf("PWM输出中，10秒后关闭...\n");
    sleep(10);

    /* 6. 禁用 PWM */
    printf("5. 禁用PWM...\n");
    ret = pwm_write(path, "0");
    if (ret < 0) return -1;
    printf("PWM已禁用\n");

    /* 7. 取消导出（可选） */
    printf("6. 取消导出PWM通道...\n");
    snprintf(path, sizeof(path), "%s/unexport", PWM_PATH);
    pwm_write(path, "1");
    printf("PWM通道已取消导出\n");

    return 0;
}
```

10.3.4 运行方式说明

1. **保存源码**：将上述完整用户态程序保存为文件，例如 pwm_control.c。
2. **编译**：在宿主机或目标板上使用对应工具链编译。示例（目标板为 ARM 时请替换为实际交叉编译器）：

```
# 交叉编译示例（以实际工具链为准）
arm-none-linux-uclibcgnueabihf-gcc -o pwm_control pwm_control.c -static
```

3. **运行**：将可执行文件拷贝到开发板后，在目标板上执行。需对 /sys/class/pwm 有写权限（通常需 root 或相应权限）。

```
./pwm_control
```

   程序将依次：导出 pwmchip6 的 pwm0、设置 1 kHz 周期与 30%% 占空比、启用 PWM 输出、运行 10 秒、禁用输出、取消导出。可在对应引脚用示波器观察波形。

4. **修改通道与路径**：若使用其他控制器或通道，请修改源码中的 PWM_PATH（如 /sys/class/pwm/pwmchip0）、PWM_NUM（如 1），以及 export/unexport 中写入的通道号，使与本文「硬件与 sysfs 对应关系」一致。

11 ADC 操作指南

本文档说明在 CV184x 平台上通过 IIO 接口使用 SARADC、通道与 sysfs 节点对应关系、电压换算及用户态/内核态操作示例。

11.1 操作准备

1. 内核要求

- 使用 SDK 提供的内核镜像。

2. ADC 与通道说明

- 用户层通过 IIO 接口访问 15 路 12-bit ADC 通道，参考电压 1.5V。
- 芯片内共有 5 组 SARADC，每组 3 路 12-bit 通道：
  - SARADC0 ~ SARADC2：常规 ADC（共 9 路）
  - RTC_SARADC0 ~ RTC_SARADC1：RTC 域 ADC（共 6 路）

3. IIO 设备路径与通道对应关系

IIO 设备路径为 /sys/bus/iio/devices/iio:device0/，各通道对应的原始值节点如下：

```
in_voltage1_raw ~ in_voltage3_raw   # SARADC0
in_voltage4_raw ~ in_voltage6_raw   # SARADC1
in_voltage7_raw ~ in_voltage9_raw   # SARADC2
in_voltage10_raw ~ in_voltage12_raw # RTC_SARADC0
in_voltage13_raw ~ in_voltage15_raw # RTC_SARADC1
```

4. 电压计算公式

电压（mV）= 原始值 × 1500 / 4095

5. 引脚与功能切换

- SARADC0 通道 3 对应的引脚名称为 ADC3；可采用默认 GPIO 功能，无需切换：

```
cvi_pinmux -r ADC3
```

- RTC_SARADC0 通道 10 对应引脚 PWR_SEQ3；需切换为 GPIO 功能，例如：

```
cvi_pinmux -r PWR_SEQ3
cvi_pinmux -w PWR_SEQ3/PWR_GPIO_5
```

11.2 操作流程

步骤 1：硬件连接

- 确认所用 ADC 通道对应的物理引脚（可参考原理图）。
- 将待测电压源接到对应引脚，注意输入电压不得超过参考电压 1.5V。
- 保证参考电压稳定。

步骤 2：引脚配置

- 常规 ADC 通道（SARADC0 ~ 2）一般无需额外 pinmux。
- RTC 域 ADC（RTC_SARADC0/1）或需将对应引脚配置为 ADC 功能，或需切回 GPIO 时，使用 cvi_pinmux 进行切换。

步骤 3：数据读取

- 用户态：通过 /sys/bus/iio/devices/iio:device0/ 下对应的 in_voltageN_raw 节点读取原始值。

步骤 4：数据处理

- 读取原始值后，用公式「电压（mV）= 原始值 × 1500 / 4095」换算为电压。
- 按需做滤波、校准等后处理。

11.3 操作示例

11.3.1 读取 SARADC0 通道 1 并换算电压

```
# 读取原始值
cat /sys/bus/iio/devices/iio:device0/in_voltage1_raw
# 示例输出：2048
# 电压（mV）= 2048 × 1500 / 4095 ≈ 750
```

说明：开发板原理图中 ADC 常按通道标号（如 SARADC1 ~ 9 对应上述 SARADC0 ~ 2 的 9 路，PWR_SARADC1 ~ 6 对应 RTC_SARADC0/1 的 6 路），具体以原理图为准。

11.3.2 SARADC0 通道 3 与 RTC_SARADC0 通道 10 读取示例

可用万用表对比：将通道接 1.8V 或接地（注意原始值最大为 4095，换算电压最大为 1.5V）。

```
cat /sys/bus/iio/devices/iio:device0/in_voltage3_raw
cat /sys/bus/iio/devices/iio:device0/in_voltage10_raw
# 接地时读数应接近 0
```

```
[root@cvitek]/mnt/sd# cat /sys/bus/iio/devices/iio:device0/in_voltage3_raw
4095
[root@cvitek]/mnt/sd# cat /sys/bus/iio/devices/iio:device0/in_voltage3_raw
0
[root@cvitek]/mnt/sd#
```

11.3.3 内核态 ADC 读写示例

步骤 1：在设备树中为需要配置 ADC 通道的节点添加 ADC 通道属性

```
my-adc-consumer@0 {
    compatible = "myco,my-adc-consumer";
    io-channels = <&saradc0 0>;  /* 0 表示通道 0，范围 0 ~ 14 */
    status = "okay";
};
```

步骤 2：在驱动中申请 ADC 通道并读取 ADC 值

```
struct my_adc_consumer {
    ...
    struct iio_channel *chan;
    ...
};

static int my_adc_consumer_probe(struct platform_device *pdev)
{
    struct my_adc_consumer *priv;
    int ret, val;

    priv = devm_kzalloc(&pdev->dev, sizeof(*priv), GFP_KERNEL);
    if (!priv)
        return -ENOMEM;

    /* 获取 ADC 通道 */
    priv->chan = devm_iio_channel_get(&pdev->dev, NULL);
    if (IS_ERR(priv->chan)) {
        dev_err(&pdev->dev, "iio_channel_get failed: %ld\n", PTR_ERR(priv->chan));
        return PTR_ERR(priv->chan);
    }
    /* 读取原始值 */
    ret = iio_read_channel_raw(priv->chan, &val);
    if (ret < 0) {
        dev_err(&pdev->dev, "read raw failed: %d\n", ret);
        return ret;
    }
    dev_info(&pdev->dev, "ADC raw value: %d\n", val);
    ...
}
```

11.3.4 用户态 ADC 读写示例

以下示例通过 IIO sysfs 读取指定通道的原始值并换算为电压（mV）。通道 1~15 对应 in_voltage1_raw``~``in_voltage15_raw，电压公式：电压（mV）= 原始值 × 1500 / 4095。用法：./程序名 < 通道号>，通道号 1~15。

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

#define ADC_PATH "/sys/bus/iio/devices/iio:device0/"
#define MAX_CHANNELS 15

float read_adc_voltage(int channel)
{
    char path[256];
    char buf[32];
    int fd, raw;
    float voltage;

    if (channel < 1 || channel > MAX_CHANNELS) {
        fprintf(stderr, "Invalid channel: %d (1-%d)\n", channel, MAX_CHANNELS);
        return -1.0;
    }

    snprintf(path, sizeof(path), "%sin_voltage%d_raw", ADC_PATH, channel);

    fd = open(path, O_RDONLY);
    if (fd < 0) {
        fprintf(stderr, "Failed to open %s: %s\n", path, strerror(errno));
        return -1.0;
    }

    memset(buf, 0, sizeof(buf));
    if (read(fd, buf, sizeof(buf) - 1) < 0) {
        fprintf(stderr, "Failed to read %s: %s\n", path, strerror(errno));
        close(fd);
        return -1.0;
    }

    close(fd);

    raw = atoi(buf);
    voltage = (raw * 1500.0) / 4095.0;

    return voltage;
}

int main(int argc, char *argv[])
{
    int channel;
    float voltage;
```

（下页继续）

(续上页)

```
50
51      if (argc != 2) {
52          printf("Usage: %s <channel>\n", argv[0]);
53          printf("Channels: 1-15\n");
54          printf("  1-3: SARADC0\n");
55          printf("  4-6: SARADC1\n");
56          printf("  7-9: SARADC2\n");
57          printf("  10-12: RTC_SARADC0\n");
58          printf("  13-15: RTC_SARADC1\n");
59          return 1;
60      }
61
62      channel = atoi(argv[1]);
63      voltage = read_adc_voltage(channel);
64
65      if (voltage >= 0) {
66          printf("Channel %d: %.2f mV (%.3f V)\n",
67                 channel, voltage, voltage / 1000.0);
68      }
69
70      return 0;
71  }
```

运行方式说明

1. **保存源码**：将上述代码保存为文件，例如 adc_read.c。
2. **编译**：在宿主机或目标板上使用对应工具链编译。示例（目标板为 ARM 时请替换为实际交叉编译器）：

```
# 交叉编译示例（以实际工具链为准）
arm-none-linux-uclibcgnueabihf-gcc -o adc_read adc_read.c -static
```

3. **运行**：在目标板上执行，传入通道号 1 ~ 15。无参数或参数错误时会打印用法与通道对应关系。

```
./adc_read <通道号>
```

示例：

```
./adc_read 1     # 读取 SARADC0 通道 1
./adc_read 10    # 读取 RTC_SARADC0 通道 10
./adc_read       # 打印用法：Channels 1-15，及 1-3/4-6/7-9/10-12/13-15 对应关系
```

4. **输出**：成功时打印该通道电压，如 Channel 1: 750.00 mV (0.750 V)；失败时在 stderr 输出错误信息。

12 CVI PINMUX 操作指南

本文档说明在 CV184x 平台上如何配置引脚复用（Pinmux）：U-Boot 内配置方式、内核下使用 cvi_pinmux 工具，以及 ephy 引脚切换为 GPIO/SPI 的示例。

12.1 操作准备

- 使用 SDK 提供的内核；若需在系统启动后使用 cvi_pinmux 工具，需在 SDK 编译前通过 menuconfig 勾选 Pinmux 相关选项，编译并烧录后即可在终端或 SSH 下执行 cvi_pinmux。

12.2 操作流程

12.2.1 U-Boot 下 Pinmux 配置

U-Boot 阶段的 pinmux 在源码文件 u-boot-2021.10/board/cvitek/cv184x/board.c 中通过宏配置，例如：

```
#define PINMUX_CONFIG(PIN_NAME, FUNC_NAME) \
  mmio_clrsetbits_32(PINMUX_BASE + FMUX_GPIO_FUNCSEL_##PIN_NAME, \
    FMUX_GPIO_FUNCSEL_##PIN_NAME##_MASK << FMUX_GPIO_FUNCSEL_#
↪#PIN_NAME##_OFFSET, \
    PIN_NAME##__##FUNC_NAME)
```

参数含义：

- **PIN_NAME**：引脚名，与 pinlist 文档中第一列 Signal Name 对应。
- **FUNC_NAME**：功能名，与 pinlist 文档中要切换到的功能（Function 列）对应。

示例：将 UART2 的 TX/RX 引脚分别配置为 UART4 的 TX/RX 功能，在 board.c 中添加：

```
PINMUX_CONFIG(UART2_TX, UART4_TX);
PINMUX_CONFIG(UART2_RX, UART4_RX);
```

某引脚支持哪些 FUNC_NAME，可查阅 pinlist 文档中该 Signal Name 对应的 Function 列。

12.2.2 内核下 Pinmux 配置（cvi_pinmux 工具）

工具准备：在 SDK 中执行 menuconfig，搜索并勾选 pinmux 相关选项后重新编译、烧录；系统启动后在终端或 SSH 中即可使用 cvi_pinmux。

命令说明：执行 cvi_pinmux -h 可查看帮助。常用命令如下：

```
cvi_pinmux -p          # 列出所有引脚
cvi_pinmux -l          # 列出所有引脚及其当前功能
cvi_pinmux -r pin      # 读取指定引脚的当前功能
cvi_pinmux -w pin/func # 将引脚设置为指定功能
```

示例：将 UART2 的 TX、RX 分别设为 UART4 的 TX、RX：

```
cvi_pinmux -w UART2_TX/UART4_TX
cvi_pinmux -w UART2_RX/UART4_RX
```

输出中会显示引脚名、目标功能、寄存器地址与写入值。可通过 cvi_pinmux -l 查看 SoC 上所有引脚及可选功能，或通过 cvi_pinmux -r pin_name 查看某引脚当前功能及可选功能。

12.3 ephy 引脚切换为 GPIO 或 SPI

以下以 ephy 相关引脚为例，说明如何将其切换为 GPIO 或 SPI 功能并做简单验证。

12.3.1 切换为 GPIO

步骤 1：通过寄存器开启 GPIO 功能

```
devmem 0x03009804 32 0x1
devmem 0x0300907C 32 0x0500
devmem 0x03009078 32 0x1000
devmem 0x03009050 32 0x4000
devmem 0x0300907C 32 0x0
devmem 0x03009804 32 0x0

devmem 0x03009804 32 0x1
devmem 0x03009808 32 0x181
devmem 0x03009800 32 0x0905
devmem 0x0300907c 32 0x0500
devmem 0x03009078 32 0x1f00
devmem 0x03009074 32 0x606
devmem 0x03009070 32 0x606
```

步骤 2：用 cvi_pinmux 将引脚设为 GPIO（以 PAD_ETH_TXP 为例）

```
cvi_pinmux -r PAD_ETH_TXP
cvi_pinmux -w PAD_ETH_TXP/XGPIOB_25
```

```
[root@cvitek]/mnt/sd# cvi_pinmux -r PAD_ETH_TXP
PAD_ETH_TXP function:
[ ] UART3_RX
[ ] IIC1_SCL
[ ] XGPIOB_25
[ ] PWM_13
[ ] CAM_MCLK0
[v] SPI1_SDO
[ ] IIS2_LRCK

register: 0x3001128
value: 6
[root@cvitek]/mnt/sd# cvi_pinmux -w PAD_ETH_TXP/XGPIOB_25
pin PAD_ETH_TXP
func XGPIOB_25
register: 3001128
value: 3
```

步骤 3：验证 GPIO 输出

GPIOB 组基号为 448，PAD_ETH_TXP 对应 XGPIOB_25，则 GPIO 编号为 448 + 25 = 473。在 sysfs 中导出并拉高/拉低，观察电平变化是否正常：

```
echo 473 > /sys/class/gpio/export
echo out > /sys/class/gpio/gpio473/direction
echo 1 > /sys/class/gpio/gpio473/value
cat /sys/class/gpio/gpio473/value
echo 0 > /sys/class/gpio/gpio473/value
cat /sys/class/gpio/gpio473/value
```

```
[root@cvitek]/mnt/sd# echo 1 > /sys/class/gpio/gpio473/value
[root@cvitek]/mnt/sd# cat /sys/class/gpio/gpio473/value
1
[root@cvitek]/mnt/sd# echo 0 > /sys/class/gpio/gpio473/value
[root@cvitek]/mnt/sd# cat /sys/class/gpio/gpio473/value
0
```

12.3.2 切换为 SPI

以 PAD_ETH_TXP、PAD_ETH_TXM 为例，将引脚切到 SPI1 的 SDO、SDI 后，可短接进行环回测试。

步骤 1：将引脚配置为 SPI1 功能

```
cvi_pinmux -r PAD_ETH_TXM
cvi_pinmux -w PAD_ETH_TXM/SPI1_SDI
cvi_pinmux -r PAD_ETH_TXP
cvi_pinmux -w PAD_ETH_TXP/SPI1_SDO
```

步骤 2：短接 SPI 的 MOSI 与 MISO，做环回测试

若 data.bin 与 data_out.bin 一致，则说明 SPI 收发正常：

```
dd if=/dev/urandom of=data.bin bs=2k count=1
./spidev_test -D /dev/spidev1.0 -s 10000000 -i data.bin -o data_out.bin
diff data.bin data_out.bin
```

13 DMA 操作指南

本文档说明在 CV184x 平台上通过 /dev/dma_memcpy 字符设备进行 DMA 内存到内存拷贝的配置、编译、加载及用户态测试方法。

13.1 dev-to-mem 操作指南

使用 DMA 内存到内存拷贝字符设备前需满足以下条件：

13.1.1 内核要求

- 使用官方发布的 SDK 编译后的内核镜像，并按下文 **mem-to-mem 操作指南**中步骤完成内核配置与编译，将 DMA 内存拷贝字符设备编为内核模块（dma-memcpy-dev.ko ）。

13.1.2 驱动说明

- 驱动为 misc 字符设备，设备节点为 /dev/dma_memcpy （加载模块后由 misc 子系统自动创建）。
- 驱动会分配 **两倍** dma_memcpy_buf_size 的 DMA 连续内存；嵌入式板卡内存有限，**请勿**将 dma_memcpy_buf_size 设为 3MB 以免 OOM。默认 1MB 即可；若需测试约 2MB 传输，可尝试模块参数 dma_memcpy_buf_size=2097152 。

13.2 mem-to-mem 操作指南

13.2.1 配置并打开内核 menuconfig

在项目根目录下执行：

```
source build/envsetup_soc.sh
defconfig <板子名>              # 例如 cv1842hp_wevb_0014a_spinor
menuconfig_kernel              # 打开内核 menuconfig
```

在 menuconfig 中：**Device Drivers** → **Misc devices** → **DMA memory-to-memory copy character device** 选 **M**，保存退出。

13.2.2 编译内核（含模块）

```
# 同上 source + defconfig 后
build_kernel
```

模块会随内核一起编出。

模块位于 linux_5.10/<out_dir>/drivers/misc/dma-memcpy-dev.ko，或从 build 的 modules 安装目录拷贝。

13.2.3 加载与设备节点

- **加载**：insmod dma-memcpy-dev.ko （**建议不传参数**，使用默认 1MB 缓冲区）。
- **设备节点**：/dev/dma_memcpy 。
- **注意**：勿使用 dma_memcpy_buf_size=3145728 （3MB），易 OOM；若需测 2M，可尝试 dma_memcpy_buf_size=2097152 ，失败则用默认 1MB。

13.3 用户态测试

13.3.1 完整测试代码

示例程序 dma_memcpy_test.c 完整源码如下：

```
1  /*
2   * DMA mem-to-mem test: 数据耗时 + 有效性测试
3   * 1) 数据耗时: 1M / 2M 拷贝并打印耗时
4   * 2) 有效性测试: 源缓冲前 4 字节写 0x12345678，拷贝后检查目的前 4 字节
5   *
6   * 嵌入式板子内存紧张，请用默认 1MB 加载模块，勿设 3MB 以免 OOM:
7   *   insmod dma-memcpy-dev.ko
8   * 测 2M 可尝试: insmod dma-memcpy-dev.ko dma_memcpy_buf_size=2097152 (仍可能 OOM)
9   */
10
11 #include <stdio.h>
12 #include <stdlib.h>
13 #include <stdint.h>
14 #include <string.h>
15 #include <fcntl.h>
16 #include <unistd.h>
17 #include <sys/ioctl.h>
18 #include <sys/time.h>
19 #include <errno.h>
20
```

(下页继续)

(续上页)

```c
#define DMA_MEMCPY_DEV_PATH  "/dev/dma_memcpy"
#define DMA_MEMCPY_IOC_MAGIC 'D'
#define DMA_MEMCPY_IOC_TRANSFER      _IOWR(DMA_MEMCPY_IOC_MAGIC, 1, struct dma_memcpy_transfer)

/* 与内核驱动一致 */
struct dma_memcpy_transfer {
    unsigned long long src;
    unsigned long long dst;
    unsigned long long len;
    int status;
};

#define LEN_1M       (1024 * 1024)
#define LEN_2M       (2 * 1024 * 1024)
#define LEN_3M       (3 * 1024 * 1024)
#define VALIDITY_MAGIC       0x12345678

static int do_one_copy(int fd, void *src, void *dst, size_t len)
{
    struct dma_memcpy_transfer xfer;
    struct timeval tv0, tv1;
    long us;

    printf("Setup DMA memcpy: src=%p, dst=%p, len=%zu\n", src, dst, len);

    xfer.src = (unsigned long long)(uintptr_t)src;
    xfer.dst = (unsigned long long)(uintptr_t)dst;
    xfer.len = (unsigned long long)len;
    xfer.status = 0;

    gettimeofday(&tv0, NULL);
    if (ioctl(fd, DMA_MEMCPY_IOC_TRANSFER, &xfer) < 0) {
        perror("ioctl");
        return -1;
    }
    gettimeofday(&tv1, NULL);
    us = (tv1.tv_sec - tv0.tv_sec) * 1000000 + (tv1.tv_usec - tv0.tv_usec);

    if (xfer.status != 0) {
        fprintf(stderr, "Transfer failed: status=%d (len > driver buf_size?)\n", xfer.status);
        return -1;
    }
    printf("dma_memcpy cost time = %ld us\n", us);
    return 0;
}

int main(int argc, char **argv)
{
    int fd;
    void *src = NULL;
    void *dst = NULL;
    size_t max_len = LEN_3M; /* 尽量 3M，便于和参考测试一致 */
    int ret = 0;
```

(下页继续)

(续上页)

```
74
75      (void)argc;
76      (void)argv;
77
78      fd = open(DMA_MEMCPY_DEV_PATH, O_RDWR);
79      if (fd < 0) {
80              perror("open " DMA_MEMCPY_DEV_PATH);
81              fprintf(stderr, "请先加载模块: insmod dma_memcpy_dev.ko\n");
82              return 1;
83      }
84
85      src = malloc(max_len);
86      dst = malloc(max_len);
87      if (!src || !dst) {
88              perror("malloc");
89              ret = 1;
90              goto out;
91      }
92
93      /* ---- 1、数据耗时 ---- */
94      printf("1、数据耗时:\n");
95
96      memset(src, 0x5a, max_len);
97      memset(dst, 0, max_len);
98
99      if (do_one_copy(fd, src, dst, LEN_1M) < 0) {
100             ret = 1;
101             goto out;
102     }
103     if (do_one_copy(fd, src, dst, LEN_2M) < 0) {
104             fprintf(stderr, "(若 2M 失败，可尝试: insmod dma_memcpy_dev.ko dma_memcpy_buf_
↪size=2097152)\n");
105             /* 不直接退出，继续有效性测试 */
106     }
107
108     /* ---- 2、有效性测试 ---- */
109     printf("2、有效性测试:\n");
110
111     memset(src, 0xff, max_len);
112     memset(dst, 0xff, max_len);
113     *(uint32_t *)src = (uint32_t)VALIDITY_MAGIC;
114
115     /* 优先 3M，否则 1M */
116     if (do_one_copy(fd, src, dst, LEN_3M) < 0) {
117             if (do_one_copy(fd, src, dst, LEN_1M) < 0) {
118                     ret = 1;
119                     goto out;
120             }
121     }
122
123     if (*(uint32_t *)dst == (uint32_t)VALIDITY_MAGIC)
124             printf("Validity OK: dst[0..3] = 0x%08x (expected 0x%08x)\n",
125                    *(uint32_t *)dst, VALIDITY_MAGIC);
126     else {
```

(下页继续)

(续上页)

```
            printf("Validity FAIL: dst[0..3] = 0x%08x (expected 0x%08x)\n",
                   *(uint32_t *)dst, VALIDITY_MAGIC);
            ret = 1;
    }

out:
    if (src) free(src);
    if (dst) free(dst);
    close(fd);
    return ret;
}
```

13.3.2 编译说明

在主机上使用 cvitek 交叉编译链、静态链接生成可在板子上运行的二进制（无需板子上的动态库）。将上文中的源码保存为 dma_memcpy_test.c 后执行：

```
arm-none-linux-uclibcgnueabihf-gcc -o dma_memcpy_test dma_memcpy_test.c -static
```

将生成的 dma_memcpy_test 拷到板子（如 /mnt/sd/ ）后执行：

```
./dma_memcpy_test
```

13.3.3 测试内容说明

dma_memcpy_test.c 主要做两件事：

1. **数据耗时**：1M/2M 拷贝并打印耗时。
2. **有效性测试**：源缓冲前 4 字节写 0x12345678 ，拷贝后检查目的缓冲前 4 字节是否一致。

13.4 ioctl 接口

- **命令**：DMA_MEMCPY_IOC_TRANSFER
- **参数**：struct dma_memcpy_transfer { src, dst, len; status; } - src / dst ： 用户空间源/目的地址 - len ： 拷贝长度（不超过模块参数 dma_memcpy_buf_size ） - status ： 返回 0 成功，负数为 errno

数据路径：用户 src → 内核缓冲 → DMA 拷贝 → 内核缓冲 → 用户 dst。

13.5 如何验证为 DMA mem-to-mem

13.5.1 看内核日志

驱动在每次 DMA 完成后会打印 dma_memcpy cost time 和 DMA mem-to-mem done 。在板子上按顺序执行：

```
dmesg -c
./dma_memcpy_test
dmesg | grep -i dma
```

若看到类似下面输出，说明走的是硬件 DMA mem-to-mem，且 src_dma/dst_dma 为 DMA 总线地址：

- dma_memcpy cost time = xxxx us
- DMA mem-to-mem done: 1048576 bytes (src_dma=0x... dst_dma=0x...)

13.5.2 看 CPU 占用

做一次较大、连续多次的拷贝，同时用 top 看进程 CPU。DMA 拷贝时 CPU 占用应很低；若改成纯软件 memcpy，CPU 会明显升高。

14 温控策略与测试操作指南

本文档说明在 CV184x 平台上查看 CPU/TPU 时钟与温度、温控策略与 trip 对应关系、迟滞（hysteresis）含义、模拟温度测试步骤，以及在设备树中修改温控阈值与迟滞的方法。

14.1 操作准备

1. 内核要求

- 使用 SDK 提供的内核，并确保内核配置中已开启 CONFIG_THERMAL_EMULATION=y（进行模拟温度测试时需开启，否则 emul_temp 写入无效，不会驱动温控逻辑）。

14.2 操作流程

14.2.1 查看 CPU/TPU 时钟

```
cat /sys/kernel/debug/clk/clk_summary | grep -E '\b(clk_cpu|clk_tpu)\b'
```

会显示 CPU 和 TPU 的当前频率。

14.2.2 查看当前温度

```
cat /sys/class/thermal/thermal_zone0/temp
```

单位为 **millicelsius**（毫摄氏度），例如 90000 表示 90°C。

14.2.3 温控策略（与 trip 对应关系）

温控通过 **thermal zone 的 trip** 与 **cooling device 档位**绑定；档位对应的 **CPU/TPU 频率**由 cv184x_cooling 节点的 dev-freqs 决定（该节点定义于 build/boards/default/dts/cv184x_arm/cv184x_base_arm.dtsi）。

表 14.1: 温度区间与频率、说明

温度区间	CPU 频率 (Hz)	TPU 频率 (Hz)	说明
tmp < 90°C	1000000000	500000000	未触发第一档 passive trip，cooling state 0
90 ⩽ tmp < 100°C	500000000	375000000	触发 trip_0，cooling state 1
100 ⩽ tmp < 120°C	500000000	300000000	触发 trip_1，cooling state 2
tmp ⩾ 120°C	—	—	**强制关机**（critical trip）

若板级或 SoC 的 dev-freqs 与上表不完全一致，以 build/boards/default/dts/cv184x_arm/cv184x_base_arm.dtsi 中 cv184x_cooling { dev-freqs = ... } 及实际 clk_summary 为准。默认 ARM 侧 DTS 常见为三档：(1100M,750M) / (500M,375M) / (500M,300M)，与 cooling state 0/1/2 一一对应。

14.2.4 迟滞（hysteresis）说明

hysteresis 用于 **降温升频**时不立刻在阈值处反复横跳，减少频率抖动。

- 单位与 temperature 相同，为 **millicelsius**。
- 典型配置：**5000** 即 **5°C** 迟滞。
- **升档（降频）**：温度 ⩾ trip 的 temperature 时进入该档。
- **降档（升频）**：温度需 **低于** temperature - hysteresis 才会退出该档。

示例（第一档 trip 90°C、迟滞 5°C）：

- 从 91°C 降到 89°C：**仍保持**降频档（未低于 90-5=85°C，不升频）。
- 继续降到 84°C：**低于 85°C** 后退出第一档，频率恢复到高档。

第二档同理：若 trip 为 100°C、迟滞 5°C，则需降到 **95°C 以下**才会从第二档回到第一档。

14.3 操作示例

14.3.1 模拟温度测试步骤

14.3.2 基线

```
cat /sys/class/thermal/thermal_zone0/temp
cat /sys/kernel/debug/clk/clk_summary | grep -E '\b(clk_cpu|clk_tpu)\b'
```

14.3.3 模拟到 90°C（应进入第一档降频）

```
echo 90000 > /sys/class/thermal/thermal_zone0/emul_temp
cat /sys/kernel/debug/clk/clk_summary | grep -E '\b(clk_cpu|clk_tpu)\b'
```

14.3.4 模拟到 100°C（应进入第二档降频）

```
echo 100000 > /sys/class/thermal/thermal_zone0/emul_temp
cat /sys/kernel/debug/clk/clk_summary | grep -E '\b(clk_cpu|clk_tpu)\b'
```

14.3.5 模拟降到 95°C（有迟滞，频率应保持第二档不变）

```
echo 95000 > /sys/class/thermal/thermal_zone0/emul_temp
cat /sys/kernel/debug/clk/clk_summary | grep -E '\b(clk_cpu|clk_tpu)\b'
```

14.3.6 模拟降到 94°C（低于迟滞退档边界，应升回第一档或高档，视策略而定）

```
echo 94000 > /sys/class/thermal/thermal_zone0/emul_temp
cat /sys/kernel/debug/clk/clk_summary | grep -E '\b(clk_cpu|clk_tpu)\b'
```

若以 100°C/5°C 迟滞为准，需低于 **95°C** 才退出第二档；具体以当前 DTS 中 soc_thermal_trip_1 的 temperature 与 hysteresis 为准。

14.3.7 模拟到 120°C（触发 critical，强制关机）

```
echo 120000 > /sys/class/thermal/thermal_zone0/emul_temp
```

设备将按内核 thermal critical 行为执行关机或复位，**请勿**在生产环境随意执行。

14.3.8 在设备树中修改温控阈值、关机温度与迟滞

默认定义在 build/boards/default/dts/cv184x/cv184x_base.dtsi 的 thermal-zones → soc_thermal_0 → trips 中：

表 14.2: trip 节点默认值

节点标签	默认 temperature	默认 hysteresis	type	作用
soc_thermal_trip_0	90000 (90°C)	5000 (5°C)	passive	第一档降频 trip
soc_thermal_trip_1	100000 (100°C)	5000 (5°C)	passive	第二档降频 trip
soc_thermal_crtical_0	120000 (120°C)	0	**critical**	**关机/临界温度**

不推荐直接改默认基线 DTS，以免影响其它板型：

- build/boards/default/dts/cv184x/cv184x_base.dtsi

应在 **板级 DTS** 里覆盖同名 trip 属性（板名替换为实际板卡名）：

- build/boards/cv184x/板名/dts_arm/板名.dts
- 使用 &label 进行 **覆盖**。

14.3.9 板级覆盖示例

在对应板子的板级 DTS（路径一般为 build/boards/cv184x/板名/dts_arm/板名.dts）末尾增加：

```
/ {

};

/* 覆盖 base 中 thermal trip：temperature/hysteresis 均为 millicelsius */

&soc_thermal_trip_0 {
    temperature = <90000>;  /* 第一档触发温度，例如 90°C */
    hysteresis = <5000>;    /* 5°C 迟滞，降到此 trip 以下 5°C 才升频退档 */
};

&soc_thermal_trip_1 {
    temperature = <100000>; /* 第二档触发温度，例如 100°C */
    hysteresis = <5000>;
};
```

（下页继续）

(续上页)

```
&soc_thermal_crtical_0 {
    temperature = <120000>; /* shutdown 温度，例如 120°C；type 保持 critical */
    hysteresis = <0>;
};
```

修改后重新编译设备树并更新镜像，cooling-maps 仍指向同一 trip 标签，**无需改 map**，仅 trip 阈值变化。

14.4 相关文件索引

表 14.3: 相关文件与路径

文件路径	说明
build/boards/default/dts/cv184x/cv184x_base.dtsi	默认 thermal-zones、trip、cooling-maps
build/boards/default/dts/cv184x_arm/cv184x_base_arm.dtsi	cv184x_cooling 与 dev-freqs
linux_5.10/drivers/soc/cvitek/clock_cooling/cv184x/clock_cooling.c	cooling 档位与 clk_set_rate 实现
板级 build/boards/cv184x/板名/dts_arm/板名.dts	用 &soc_thermal_trip_0、&soc_thermal_trip_1、&soc_thermal_crtical_0 覆盖阈值